AFRL-IF-RS-TR-2002-284
Final Technical Report
October 2002

INDEPENDENT MONITORING FOR NETWORK SURVIVABILITY

Telcordia Technologies

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** October 2002	**3. REPORT TYPE AND DATES COVERED** Final Mar 97 – Aug 02

4. TITLE AND SUBTITLE INDEPENDENT MONITORING FOR NETWORK SURVIVABILITY	**5. FUNDING NUMBERS** C - F30602-97-C-0188 PE - 62301E PR - F253 TA - 40 WU - 26
6. AUTHOR(S) Mark Garrett, Bruce Siegell, Joseph Desmarais, David Shallcross, Jason Baron, Paul Seymour, Fan Chung Graham, and Christian Huitema	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Telcordia Technologies 445 South Street Morristown New Jersey 07960	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFG 525 Brooks Road Rome NY 13441-4505 Air Force Research Laboratory/SNHS 31 Grenier Street Hanscom AFB MA 01731-2909	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2002-284

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Luigi Spagnuolo/SNHS/(781) 377-4249/ Luigi.Spagnuolo@hanscom.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

This project addresses the problem of inferential topology discovery and network performance assessment. Specifically, we invent and analyze algorithms to deduce a network's internal topological structure and performance from delay and loss data based on packets exchanged among a set of monitors. The method assumes independent of network management, i.e., with no querying of routers or routing protocol data. This is a new and deep research area, especially difficult because, as we show, end-to-end measurement data is fundamentally under-determined to solve for the network topology (in straightforward manner). Among our methods, we have several negative results, and some successes, that make significant progress toward a general solution. A prototype monitoring system has been constructed and used to collect network data. The measurement technique is "sparse active monitoring" where monitors create their own traffic, but at a very low level. We address related issues of the monitor design, data collection and storage, web-based graphical user interface for analysis, and rendering of network maps. We found that our monitors were not sufficiently well synchronized to make accurate one-way delay measurements, and we discuss a method for correcting the delay data for clock drift.

14. SUBJECT TERMS Intrusion Detection, Network Monitoring, Inferential Topology Discovery, Network Performance Assessment	**15. NUMBER OF PAGES** 106 **16. PRICE CODE**

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

LIST OF FIGURES AND TABLES

List of Tables

1. Introduction

The Felix Project's goal is to research and prototype a network health monitoring system that is independent of traditional network management infrastructure. This independence provides several advantages that can contribute to more robust network management generally, and allow network monitoring when normal network management is absent or compromised. There are two "conventional" approaches to keep track of network topology. The oldest, and still pervasive, method is to manually construct a database as the network is built. Network topology data in commercial telephone and internet service providers is notoriously inaccurate due to human error in tracking changes as the network evolves. A more sophisticated method, which is now emerging in standards and practice, is network *autodiscovery*, where a network management system finds and tracks the identity and location of network elements through a protocol such as SNMP [SNMP]. Autodiscovery places some level of processing load on the network elements and creates excess traffic in the network. It also requires secure access permissions between the network elements and the management system. (An exception, of course, is the internet ICMP protocol, which, if enabled, allows any internet host to query routers for their identity and a timestamp.)

Autodiscovery is a reasonable basis for automated network management, but can still be inoperative or inadequate in some situations. First, packet networks today are composed of many technologies, particularly at layer 2. While a set of IP routers may be considered *the network* at layer 3, there can be packet congestion, failure and errors in layer 2 network elements that are not visible to layer-3 network management tools. For example, ICMP-based *ping* and *traceroute* are commonly used to identify IP routers on a path, but these programs will not see ATM, Frame Relay, MPLS, or "optical-layer" switches. A second reason autodiscovery may be insufficient is that network management mechanisms must be consistently built into all of the network elements, and that requires standardization, acceptance by multiple vendors, and time for implementation. Networks can reasonably be expected to contain technologies that are either too new or too old to be compatible with any given autodiscovery (or general network management) system. Third, the goal of network monitoring may be to evaluate a network that is not controlled by the observer. In a commercial context, the quality of an operator's service may depend on the transport provided by another operator, whose network is part of the end-to-end transport path. In a military context, the observer may wish to analyse a foreign network inconspicuously. Finally, networks may be subject to intrusion or attack (or simply software bugs) that compromise some network elements or connectivity.

Another goal, apart from topology discovery, is the assessment of network performance in terms of delay, loss, throughput or traffic load on each link of the network. The derivation of link performance is essentially a decomposition of the end-to-end behaviors measured on all paths. As we show, a straightforward equation end-to-end delay to link delay, is under-determined, so more sophisticated methods are necessary. We present some insights on the performance assessment problem, but focus mainly on topology discovery in our algorithmic work.

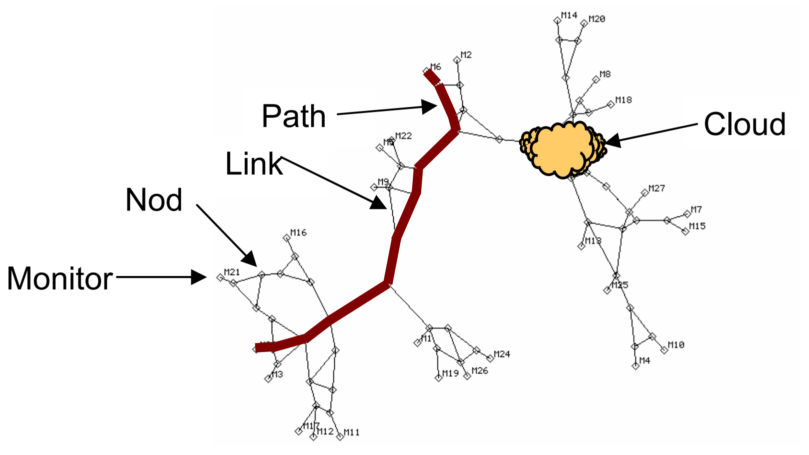


Figure 1. Example network topology graph

1.1 Approach and terminology

Figure 1 illustrates our terminology for network topology. A set of *monitors* are placed throughout the Internet (or within a regional network). The monitors measure delay and loss properties of the network by transmitting packets to each other, and recording for each packet the source, destination, starting and ending timestamps and a sequence number. Measurements are taken for every combination of source and destination monitors. The network is represented as a graph of *nodes* (e.g., routers, switches) and *links* (or *edges* in graph theory terminology). We use the term *node* to mean an interior network node, as opposed to a monitor, which of course, is also a vertex of the graph. While there may be many *routes* that connect a source and destination, the one that the packet actually takes (determined by routing protocols) is called the *path* between the two monitors. A region of the network topology that cannot be ascertained is called a *cloud*. Although do not know everything about a cloud, we may know what it is connected to, and something about the collection of links and nodes it contains.

Figure 2 shows a “big picture” view of the Felix network management problem. Measurements from monitors on a real network or from a simulation (such as the one we have developed for this project) constitute a set of raw data, with a record for each packet sent between any pair of monitors. A *topology discovery* process ascertains the structure of the network that contains the monitors and the paths interconnecting them. Topology discovery may be composed of several algorithms working together, and there may be some meaningful intermediate results that we can identify. These are especially useful if there are several ways to get to the result, and several independent methods to get from that result to the final topology. For example, a “common component matrix” identifies which paths share some network element, though we may not know anything more specific about the common elements. At a subsequent stage, we may have a “path-component matrix”, where we know which network elements (links or nodes) are on each path between monitors, but we do not know the order. Finally, when we can correctly order the elements, we have enough information to draw a graph. A variation of the topology discovery problem is where we have an initial topology, and wish to validate or correct it. For example, the system may be initialized semi-manually, and then rely on inferential methods to keep the network model up to date.

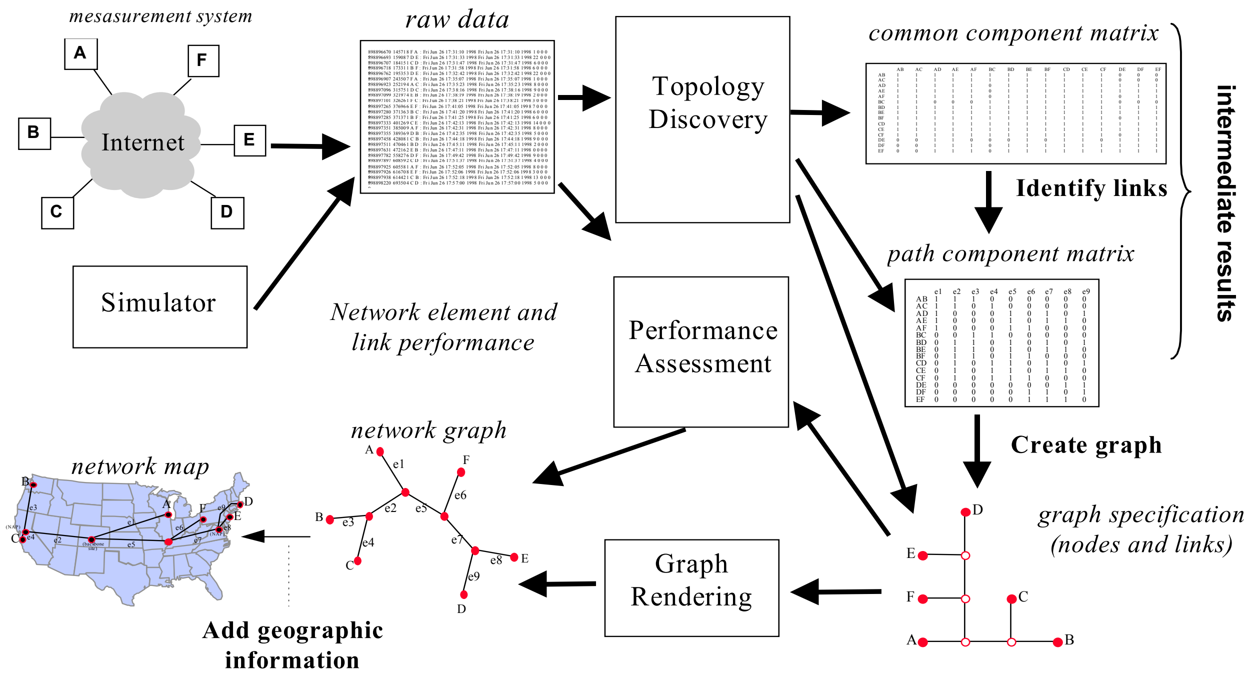


Figure 2. Overall approach to the topology discovery problem.

The *performance assessment* algorithms decompose what is known about the end-to-end performance along each path into performance information for each intermediate link or network element.

The next step in rendering a map of the network is to draw the graph in a logical and legible way. If we have some geographic information about the network, we can render the graph into a geographically meaningful map. A reasonably useful map can be produced even when we know the location of only a subset of the network elements, such as the monitors themselves. The final target output of the network management system envisioned is a network map with independently discovered topology and a quantitative assessment of the performance of the network elements. In situations where we have incomplete information about the network or its performance, the system should endeavor to render that partial information in the most meaningful and useful way.

1.2 Implications of Felix measurement methodology (some lessons learned)

The system of network monitoring that we adopt, where network properties are inferred from the behavior of probe packets has certain fundamental limitations and properties.

1.2.1 The discoverable network is a *reduced* graph

Consider a general portion of the Internet with some number of Felix monitors installed and exchanging packets. If the monitors are well distributed, most of the backbone links will be traversed by monitor packets, and can then be discovered. As we move from the backbone to the edges of the network, it becomes more likely that monitor packets will not traverse network links, because there are (presumably) many more access points in the network than there are monitors. In general, areas of a network that do not lie on a path between two Felix monitors cannot be discovered by this system. Apart from the obvious implication that the discovered network is incomplete, it is also true that a different choice of monitor locations will change the portion of the network that is discovered..

Another limitation arises when two links are traversed by identical sets of paths. If either of these links creates packet delay or loss, the monitoring system cannot know which of the links to assign the behavior to. We refer to links mapped to an identical set of paths *equivalent edges*, or, if they occur consecutively, *series edges*. If we eliminate these non-discoverable elements of the true network by deleting elements not traversed by monitor packets, and combining equivalent edges, we construct a *reduced graph* representation of the network. Although it is incomplete, the reduced graph is closely related to the actual network, and the discovery of the reduced graph may be enough information to be useful for network monitoring. Figure 3 illustrates both types of non-reduced elements in a network.

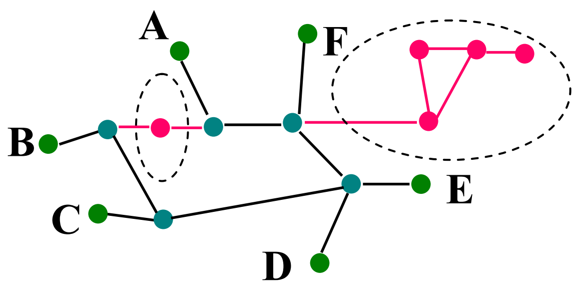


Figure 3. With Felix-type monitors, the discoverable network is a "reduced" graph, with equivalent edges and non-traversed edges (circled) removed.

1.2.2 Routing, asymmetric routes, and uni-directional links

The elimination of (non-series) equivalent edges may be a reasonable assumption because it is (roughly) consistent with how routing protocols work in the Internet. Most routing algorithms today choose between alternate routes by first minimizing the number of hops, and then breaking any ties with a consistent (non-random) local property such as port number, or next-hop IP address. We believe (without formal proof) that *equivalent* edges will not occur with this sort of rule. Note, however, that if a non-local tie-breaker is used, such as the source or destination IP address, then *equivalent* edges can occur.

Routes in the Internet are often asymmetric. That is, the path chosen from monitor A to monitor B need not include the same set of nodes as the path from B to A. This is easy to see, since the tie-breaking decision metrics in each direction are unrelated. Also, we generally assume that physical links are bi-directional, i.e. if capacity exists from node n_1 to n_2, then it is also available from n_2 to n_1. However, routing works along each path independently, so asymmetries are common.

In our network models, we generally assume that all links are uni-directional (although some broadcast local-area networks like ethernet are actually bi-directional). There are generally more powerful results in graph theory for graphs with bi-directional edges than for those with unidirectional edges, and we often had to resist the temptation to find algorithms and results for bi-directional networks (which would be too unrealistic a modeling assumption).

1.2.3 Multiple instances of a single node

For some topologies and routing of paths, a network node may appear in the discovered topology as two nodes, even when the algorithm is otherwise successful. In such cases it may be very difficult to identify the two apparent nodes as actually being the same. For example, suppose a set of paths enter and exit a node by some subset of the node's ports. Now, suppose a second set of paths enter and exit the same node through a completely disjoint set of ports. Then the topology discovery algorithm can not possibly discover that the two apparent nodes are, in fact, the same. In the context of traceroute monitoring (where we know the IP addresses of the router ports), it may sometimes be possible to identify the nodes by correlating DNS information that maps IP addresses to router names. However, each port has an individual address, and DNS entries often do not have the same name for ports on the same router.

1.2.4 Insufficient rank in path/link matrix

One of the premises of this work, at least as originally proposed, is that we can take advantage of the relationship that the path delay is the sum of link delays, or in matrix form, the vector of path delays is the product of a path-link matrix and a vector of link delays. We found this fact to be of only limited usefulness, because the rank of the path-link matrix is always insufficient to solve the inverse formula (i.e., determine link delays from path measurements). To illustrate this fact, consider adding a constant delay to each link going into a node, and subtracting the same quantity from each link going out of that node. The path delays are unchanged, implying there is an undetermined free variable in the system. In fact, there *are at least* as many degrees of freedom as there are interior nodes in the network. Therefore much more information about the system is needed to solve for the link topology and behavior.

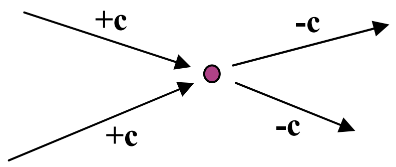


Figure 4. Link delays are not fully determined by path delays because of excess degrees of freedom. For example, a constant added and subtracted to delays of links entering and exiting each node as shown, changes link delays, but leaves path delays unchanged

1.3 Summary of results (and guide to reading this document)

The major results of the project are the construction of the prototype network monitoring system (and the lessons learned in that process), and the algorithms and methodology for topology discovery based on network delay measurements. These two activities occupied the vast majority of our time and effort. (Note, in this section, we will summarize the project in order of the significance of results, although the document is structured generally along the flow of Figure 2.)

The Felix system, shown in Figure 5, consists of network monitors that send packets to each other using a special protocol. We constructed and deployed these monitors within our corporate network (spanning several locations in New Jersey), and also in a wider area among a small set of universities. The monitors collect data and send the information to an archive server that compiles and maintains a database. The archive server performs simple statistical analysis of the delay time series data, renders graphs, and displays the results through a web-based GUI. The GUI can be used to initiate measurements, analyze data, and control the general network management system, including the components that discover network topology, assess performance, render graphs and maps, etc.

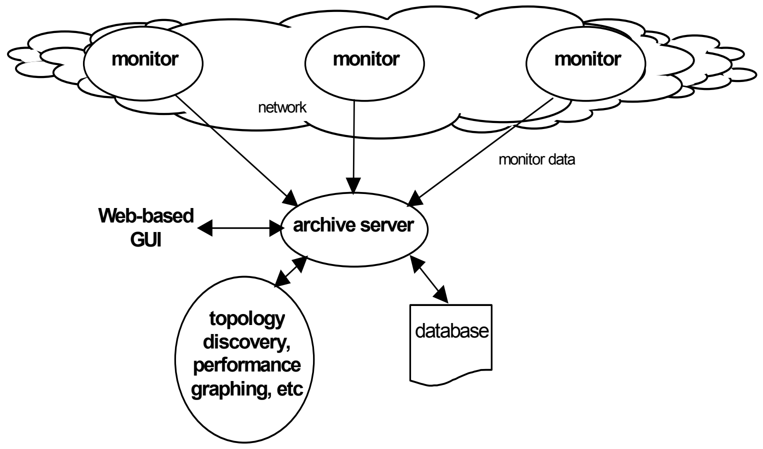


Figure 5. Basic schematic of Felix network monitoring system prototype

Section 2 presents a walk through the Felix system prototype, identifying all of the specific programs used in each demonstration, the protocols and intermediate files connecting the programs. The system, as it appears through the GUI, is intended to illustrate the network management system we envision. Many of the algorithms we have invented are not mature enough at the end of this project to be integrated together. These are displayed in the system as demonstrations.

Section 3 discusses the design of the monitor code, data collection system, database, and GUI. **Appendices A, B and C** provide detailed documentation of all of the code, protocols and file types used in the system. **Section 9** is a technical specification of the computing environment for the Felix network monitoring system.

The most significant results of the project, in terms of new theory, relate to the problem of ***network topology discovery***. **Section 6** presents the algorithms and insights we have developed in this area. We name the six methodologies as follows:

- matrix method
- tree-growing method
- spike-tail method
- correlation method
- matroid method
- distance-realization method

The *matrix* method was the original idea in the project proposal to exploit the fact that link delays sum to the path delays. This fact does not provide enough information to decompose the data into meaningful link delays because the system is provably underdetermined. One of the original ideas of the proposal, which is still useful and central to all of the methods, is that the path characteristics can be expressed as a combination of link characteristics. Specifically, link delays sum to the path delays; the loss probabilities are related as: $\log(1-P_{l\text{-path}}) = \Sigma \log(1-P_{l\text{-link}})$; the throughput along a link is the minimum across the links; and the load on a link is related to the probability that a packet traverses a path in minimum time (see the spike-tail method discussion in section 6.3). These relationships also connect together the concepts of topology discovery and link performance assessment.

The *tree-growing* method is a relatively simple algorithm to construct a tree network that introduces new links sequentially that best fit the minimum delays observed along each path. Although too limited to discover more realistic and complex topologies, we thought this exercise might be a useful tool to combine with other algorithms later (though it is still not clear how at this point).

The *spike-tail* method exploits a characteristic observed in data that we collected – namely that delays values have a heavy concentration around the minimum (propagation) delay, and then a long statistical tail where delays are spread over a large range. By estimating the proportion of delays in each region of the distribution, we can estimate the load along the path, and then decompose those metrics to yield link loads, and identify network congestion points. This method suffers from the same under-determination as the matrix method, and its solution remains incomplete. Still, we believe a useful algorithm could be derived here with further research. This method is useful for ***network performance assessment*** on the network links more than for topology discovery.

The *correlation* method works together with the *matroid* method to provide a complete solution from delay data to network graph. The basic idea of the correlation method is sound, though it still needs work on practical details. Here we compare the time-varying delays in pairs of paths and look for strong correlations that indicate congestion on a portion of the network that is shared by the two paths. The pattern of correlations across all paths can identify all links (that are sufficiently busy to show congestion events) in the network. A simulation study has been devised to test the ideas in this method and further refine the algorithms. The simulation starts with assumptions about network congestion that favor the algorithm. The assumptions are then relaxed toward a more realistic model. The initial results are promising and demonstrate the proof of concept for this method.

The *matroid* method picks up from this intermediate result. Given the list of which links are on each path, this graph-theory based algorithm constructs the network graph by first building the trees of paths terminating on each monitor, then merging them into the true network.

The *distance-realization* method starts from a distance metric, such as the minimum observed delay along each path (i.e., the propagation delay). It then realizes the graph by applying certain rules the progressively simplify the graph from a fully connected mesh to the most simplified graph that still meets the distance constraints. This algorithm is moderately successful for reasonably large graphs, but will always fail to find some specific features.

In the evaluation and demonstration of all of our topology discovery methods, we make use of some code from Georgia Tech [Cal97] that produces network topologies that structurally resemble the true Internet.

Another area where we produced some interesting results is in ***topology graph and map rendering***, described in **Section 7**. The issue is to render maps of networks when the graph or geographical information is incomplete. In the case where we have no geographical information, the graph must be rendered randomly, but in such a way that it is legible and useful. A program we created called *topomap* makes use of a well-known algorithm that anneals a graph layout into a very readable form. A modification of this algorithm allows the user to "nail down" certain nodes in the graph, and let the rest be rendered by the algorithm. We can very usefully compare an original topology to one derived from simulated data on that graph by fixing the position of the monitors. This yields two graphs that can be visually compared.

Section 7 also discusses ***graphing from traceroute data***. Traceroute identifies more information about internal network structure than Felix monitors, but the problem of drawing a meaningful graph is still an issue.

Section 4 discusses our deployment of monitors and collection of real network data. A description of the datasets and issues related to data collection are given. The most valuable lessons we learned from the data collected were that 1) clock synchronization is a major difficulty in measuring one-way delay, 2) that delay data is often heavy-tailed (which has implications in statistical analysis), and 3) that route changes in the Internet are extremely common. Once we discovered the severity of clock drift, we established a collaboration with researchers at Advanced Network & Services, Inc., who had a similar project (called "Surveyor") to measure one-way delays across the Internet. Their monitors are time-synchronized using GPS receivers, which provide adequate accuracy for such measurements. They graciously permitted us access to large sets of data collected in their system. (The focus of their work is more on evaluation of Internet path delays from one region to another, based on delay time series, while ours is mainly on inferential topology and link-performance discovery.)

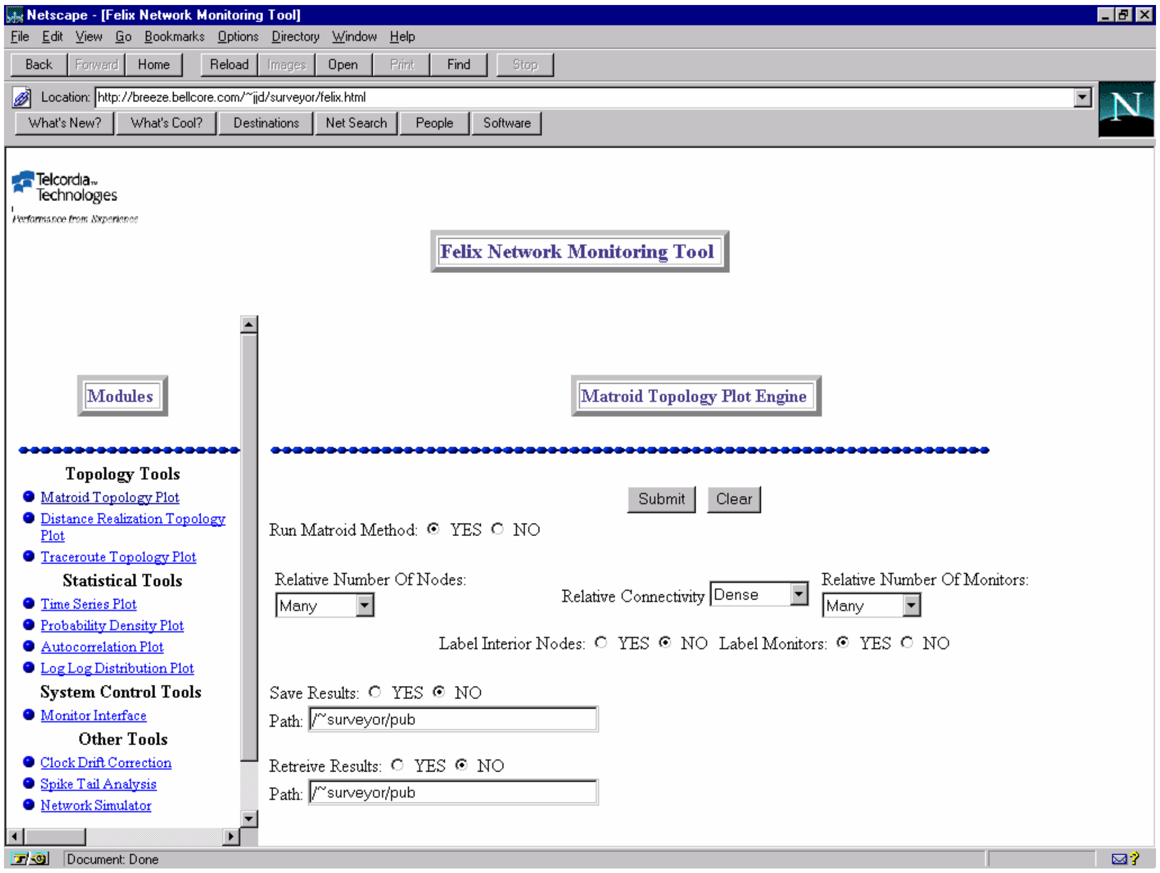
Section 5 discusses our experience with monitor clock synchronization, and presents a method to correct measured data for clock drift. Even for monitors running on workstations with NTP, we found the clock synchronization to be insufficient for one-way delay measurements.

Section 8 identifies our ideas for further research in the area of inferential topology discovery and network management. We believe this topic will be a rich source of fruitful investigation for many years. The key element that makes this field interesting and difficult is the need to derive useful intelligence from very fragmented data. The key insight of our *cross-correlation* method is that we can make use of the rich information of time-dependent behavior to compensate for the under-determined relationship between path and link properties. The clock drift correction problem that we observed has a similar flavor. That is, take whatever data the available measurements will yield, and find a way to transform and extract from that data, as much relevant intelligence as possible.

1.4 Acknowledgements

Thanks to Sumit Khurana, our summer student in 1997, for building some of the demonstrations. Also to Alex Poylisher for building the first version of the Felix simulator, and Sudha Ramesh for her work on the archive server. We are grateful to Guy Almes and Matt Zekauskas at Advanced Network & Services, Inc., for providing access to their measurements of one-way delay over the U.S. Internet. Thanks also to Will Leland for many formative discussions and guidance throughout the project.

2. A walk through the Felix Network Management System prototype


Netscape - [Felix Network Monitoring Tool]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
What's New? What's Cool? Destinations Net Search People Software
Telcordia Technologies
Performance from Experience
Document: Done

Location: http://breeze.bellcore.com/~jjd/surveyor/felix.html

Felix Network Monitoring Tool

Modules

Topology Tools
- Matroid Topology Plot
- Distance Realization Topology Plot
- Traceroute Topology Plot

Statistical Tools
- Time Series Plot
- Probability Density Plot
- Autocorrelation Plot
- Log Log Distribution Plot

System Control Tools
- Monitor Interface

Other Tools
- Clock Drift Correction
- Spike Tail Analysis
- Network Simulator

Matroid Topology Plot Engine

Submit Clear

Run Matroid Method: [✓ YES] NO

Relative Number Of Nodes: Many Relative Connectivity: Dense Relative Number Of Monitors: Many

Label Interior Nodes: YES [✓ NO] Label Monitors: [✓ YES] NO

Save Results: YES [✓ NO]
Path: /~surveyor/pub

Retreive Results: YES [✓ NO]
Path: /~surveyor/pub

Figure 6. Felix system GUI showing Matroid method demonstration in frame

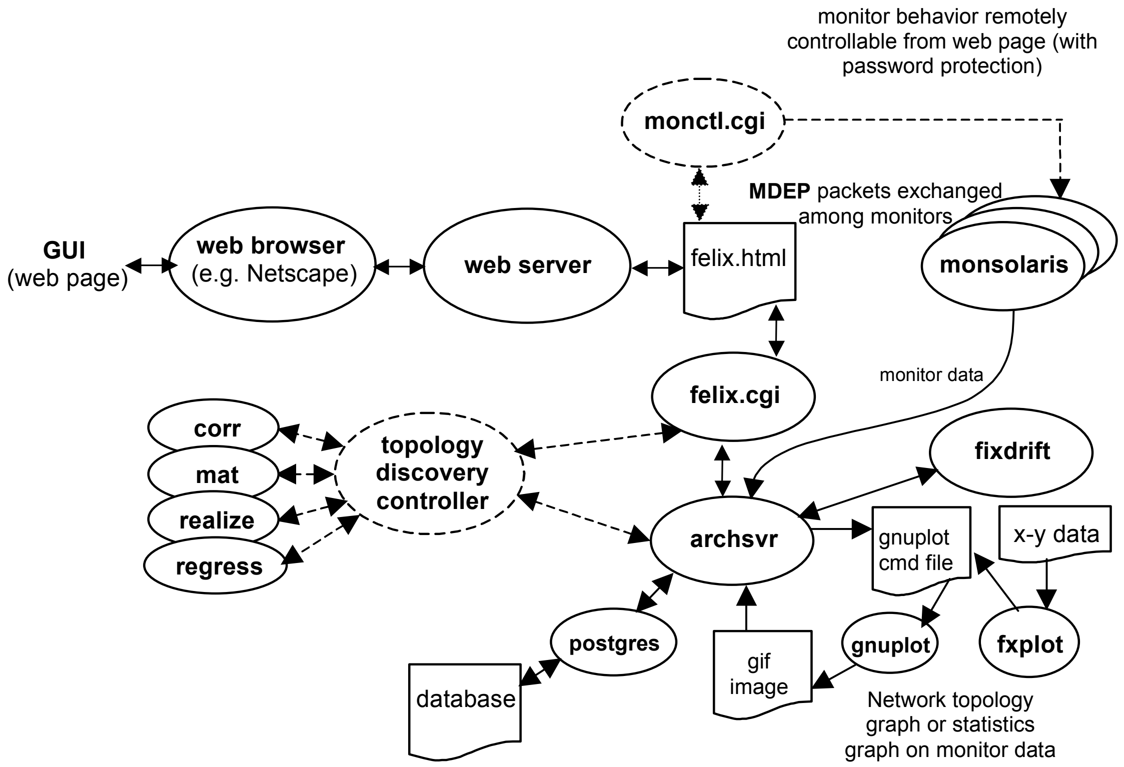
The Felix system prototype software is described here as a number of "tool chains", each of which demonstrates a set of related system capabilities. Ideally, all of the modules would be combined into a unified system. Because of the technical difficulty of inferential topology discovery, clock drift correction, and other aspects of this work, the results are given as eight separate combinations of software modules, each demonstrating the efficacy of their algorithms to solve a particular problem. The tool chain diagrams show the sequence of software components, intermediate files, and the web-based GUI (implemented as html web-pages and cgi scripts) that controls the overall operation. The seven tool chains are as follows:

- Topology discovery and statistical data analysis
- Correlation / matroid method based on simulation
- Distance realization method demonstration
- Matroid method demonstration
- Tree-growing method
- Traceroute analysis
- Clock drift correction

2.1 Topology discovery and statistical data analysis tool chain

This diagram shows how we envision the Felix prototype to operate as a network management system. Felix network monitors (*monsolaris*) send data to the "archive server" (*archsvr*) module, which stores the data in a database (implemented here using postgres). Through a web-based GUI, an operator starts at the page `felix.html`, that allows access to the data in the form of time series graphs. Various functions can be run on the data (implemented in *archsvr*) to see statistical properties such as distribution, pdf, distribution tail behavior, etc. These may be useful for an operator to investigate anomalous behavior in the network, and are also useful in the development of our algorithms. A program called *fxplot,* containing similar statistical analysis tools, is used for off-line analysis of the data (separate from the web-based demonstration system). From the `felix.html` web page, the operator can also request topology discovery on a particular set of measurements (delimited by the set of monitors and the start and ending times). A controller function would gather the data set from the archive server, pass it on to the topology discovery engine, and mediate the results from one or a collection of the algorithms. The result, which could be a network map annotated by performance data on the links, or indications of "hot spots" in the network, would then be returned to the GUI and displayed to the operator.

From the GUI, an operator could remotely control the configuration of the monitors (through *monctl.cgi*). Another function shown is *fixdrift*, that allows for correction of clock drift effects in the data before processing by topology discovery algorithms.



monitor behavior remotely controllable from web page (with password protection)
monctl.cgi
MDEP packets exchanged among monitors
GUI
(web page)
web browser
(e.g. Netscape)
web server
felix.html
monsolaris
monitor data
felix.cgi
corr
mat
realize
regress
topology discovery controller
fixdrift
archsvr
gnuplot cmd file
x-y data
postgres
gif image
gnuplot
fxplot
database
Network topology graph or statistics graph on monitor data

Figure 7. General Felix system tool chain with topology discovery and data analysis capability. Functions in dotted lines were envisioned but not implemented in this project.

Although we were successful, to varying degrees, in devising algorithms for topology discovery, they are not mature enough at the end of this project to implement as the fully integrated system envisioned here. The dotted lines indicate *envisioned* modules and interfaces that are not implemented in the Felix code at this time. The network monitoring, data collection, archiving, statistical analysis, and data graph rendering functions are implemented and operational.

2.2 Correlation / Matroid method simulation tool chain

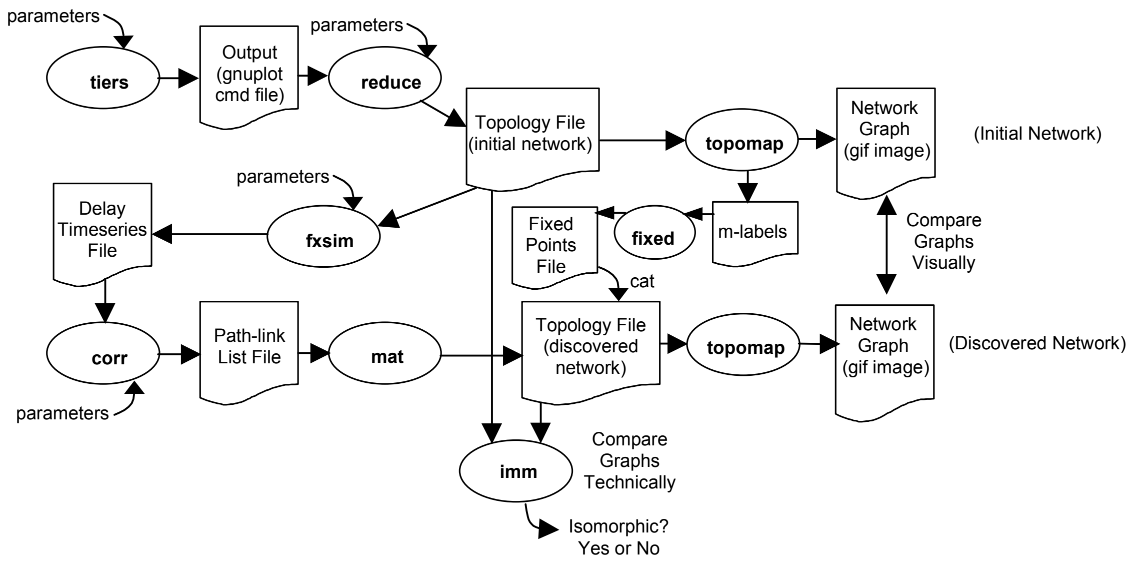


Figure 8. Tool chain showing simulation-driven topology discovery with correlation and matroid methods

The correlation method tool chain (Figure 8) demonstrates our most successful topology discovery method. The combination of the “cross-correlation” method, instantiated in the *corr* program, and the “matroid” method, in the program *mat*, is the best method we have developed that starts with packet delay data and produces a network topology graph. Currently corr works on a somewhat simplified model of network congestion. This demonstration is therefore based on simulated network delay data.

In this system, we generate an artificial graph using *tiers*, then with *reduce*, we place a specified number of monitors on leaves of the graph (randomly), and reduce that graph to a discoverable network by removing all links not traversed by monitor packets using shortest-path routing, and removing series edges. The resulting topology file can be displayed graphically using the *topomap* program. *Topomap* will render a legible network graph without any orienting information, so the placement of monitors and internal network nodes on the diagram will be random. The flow from *topomap* through the *fixed* routine creates data that is appended to the second topology file. This data records the co-ordinates of monitors in the first graph, and permits *topomap* called on the second graph to render it with the monitors in identical positions. By fixing the monitors identically in the two graphs, it is easy to compare them visually. The first topology file (with the network generated from *tiers*), is the input for *fxsim*, which simulates traffic on the network, and generates time series of delays for each path between monitors. *corr* then correlates the path observations and derives the list of links on each path. *mat* is a program that constructs a graph from that information. (The details of *corr* and *mat* are described below in the section on topology discovery algorithms.) The *imm* program checks the two topology files formally to see if they are graphically identical (isomorphic). The two graphs generated by *topomap* from the topology files may also be compared visually, which is useful for finding the difference between the two graphs (*imm* will only determine whether they are different, but not how they differ).

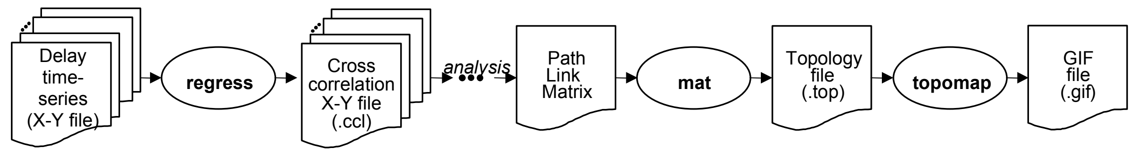


Figure 9 Correlation method tool chain with regress program (to cross-correlate time series data)

Figure 9 shows another tool chain for the correlation method that uses the *regress* program. *regress* creates the time-dependent (windowed) cross-correlation function from two time series, that can be analysed (the missing "analysis" function corresponding to *corr* above) to produce a path-link matrix to feed into *mat* as in the above tool chain. The experiments with *corr* use a simplified cross-traffic model because the results with *regress* showed that the cross-correlation of real traffic traces could not predict, in a simple way (e.g. by thresholding), that two paths did or did not share a link. At the end of the project we are still experimenting with simple traffic congestion models, with a plan to evolve the algorithms toward more realistic traffic models. (Note, Figure 15 show a tool chain that derives X-Y files from the Felix monitor database.)

2.3 Distance realization method demonstration tool chain

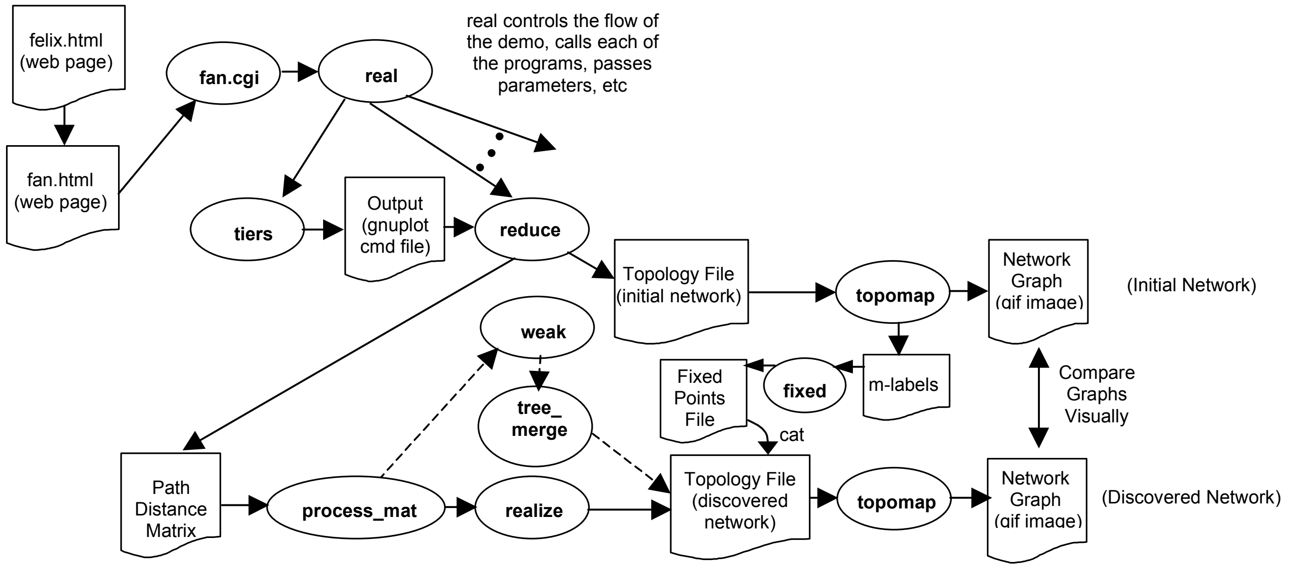
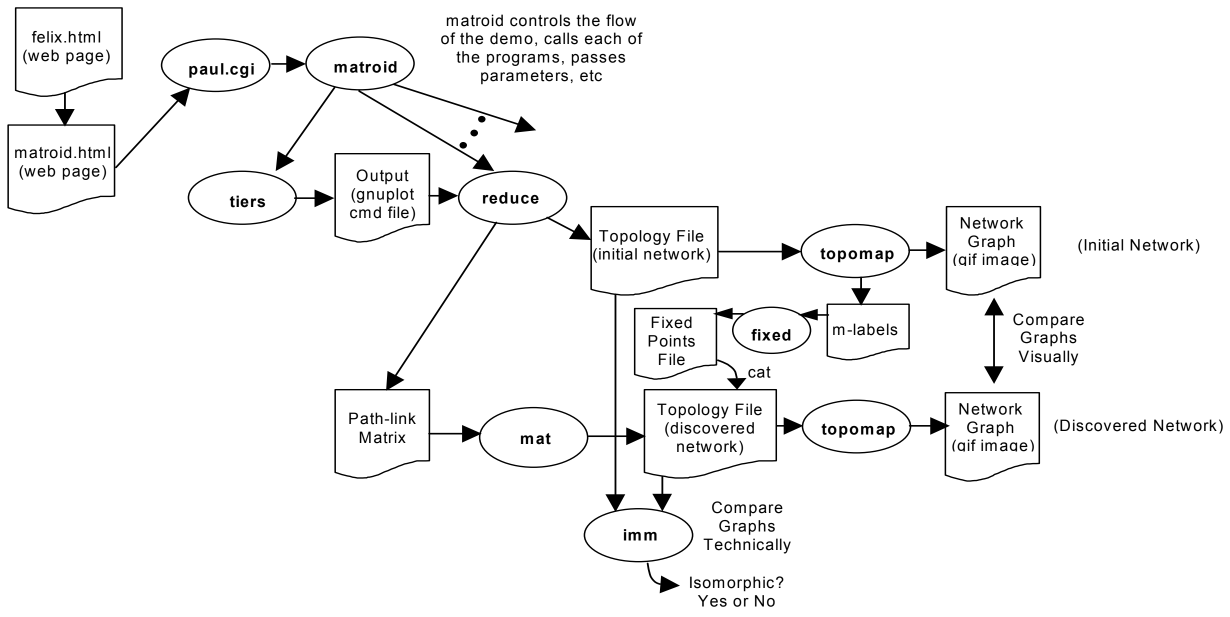

felix.html (web page)
fan.html (web page)
fan.cgi
real
real controls the flow of the demo, calls each of the programs, passes parameters, etc
tiers
Output (gnuplot cmd file)
reduce
Topology File (initial network)
topomap
Network Graph (gif image)
(Initial Network)
weak
tree_ merge
fixed
m-labels
Fixed Points File
cat
Compare Graphs Visually
Path Distance Matrix
process_mat
realize
Topology File (discovered network)
topomap
Network Graph (gif image)
(Discovered Network)

Figure 10 Tool chain for demonstration of Distance-Realization method for topology discovery.

This tool chain demonstrates another topology discovery method called "distance realization". It has several forms that are instantiated in the programs *process_mat, realize*, *weak* and *tree_merge*. The surrounding system allows a user to create a randomly-generated internet graph using *tiers*, with parameters describing the size and density of the network selected in the GUI, under interactive control from *fan.cgi*. *real* is a program that controls the execution of the subsequent programs. *tiers* and *reduce* produce a reduced topology file, and *reduce* also generates numbers for the distance (i.e., propagation delay in the network) for each path. The distance realization method programs use this information to discover the network topology. The topology files (before and after discovery) can be viewed and compared as in the correlation method described above. The program *process-mat* ensures the matrix is nonnegative, symmetric, and obeys the triangle inequality. *realize* generates a strong realization directly. Alternatively, *weak –rall* will generate rooted weak realizations that are then merged into a strong realization by the program *tree_merge*. The distance realization method for topology discovery is described in detail in subsequent sections below.

2.4 Matroid method demonstration tool chain



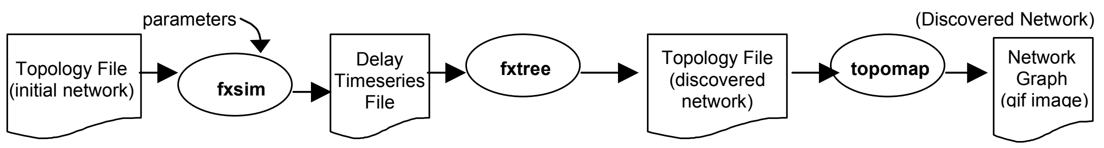
felix.html (web page)
matroid.html (web page)
paul.cgi
matroid
matroid controls the flow of the demo, calls each of the programs, passes parameters, etc
tiers
Output (gnuplot cmd file)
reduce
Topology File (initial network)
topomap
Network Graph (gif image)
(Initial Network)
m-labels
fixed
Fixed Points File
cat
Compare Graphs Visually
Path-link Matrix
mat
Topology File (discovered network)
topomap
Network Graph (gif image)
(Discovered Network)
imm
Compare Graphs Technically
Isomorphic? Yes or No

Figure 11 Tool chain for demonstration of Matroid method for topology discovery.

Similar in structure to the distance-realization tool chain, this one shows the operation of the matroid method working alone (in the *mat* program). Here *paul.cgi* controls the interactive GUI and sets parameters for the graph. *matroid* controls the flow of the demonstration. *reduce* generates the list of links on each path in the true topology, which is fed into *mat*. The rest follows as described in the previous sections.

The algorithm in *mat* outputs the graph topology that it can discover with the possibility of clouds in the output. These "clouds" indicate regions of uncertainty about how the graph is connected.

2.5 Tree-growing method tool chain



parameters
(Discovered Network)
Topology File (initial network)
fxsim
Delay Timeseries File
fxtree
Topology File (discovered network)
topomap
Network Graph (gif image)

Figure 12 Tool chain for tree-growing method for topology discovery

This method is driven by a simulation, using *fxsim*, of a network with Felix monitors. The delay time series files can also be derived from the Felix database. *fxtree* creates a tree topology that is consistent with the minimum (i.e., propagation) delay on each path in the data.

2.6 Traceroute analysis tool chain

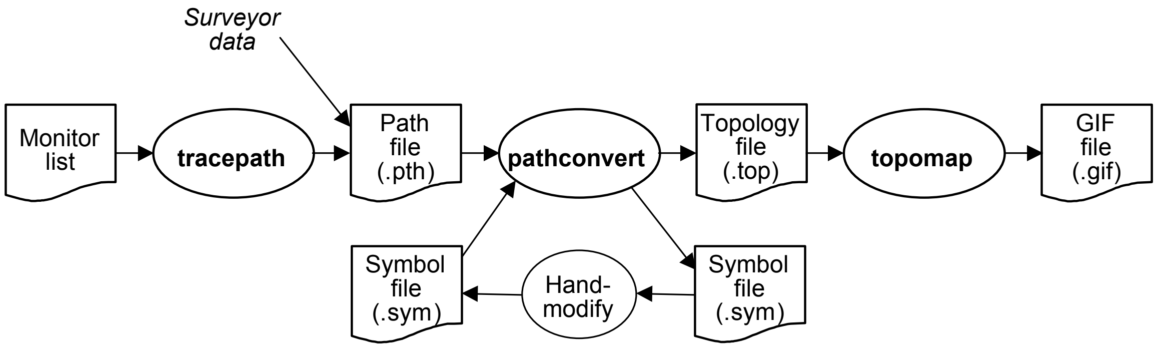


Figure 13. Traceroute tool chain: from monitor list or Surveyor data to topology graph.

Traceroute is a popular tool used to trace the route of packets in the Internet by causing routers along the path to generate an ICMP error message that identifies each router. The traceroute tool chain (shown in Figure 13) converts information returned by one or more invocations of the traceroute program into a GIF file containing a rendering of the network. The chain consists of the following steps:

1. Generate path file.

   a.) *tracepath* – run on each monitor to determine forward paths to each of the other monitors. Data from each of the monitors must be merged into a single path file. Or,

   b.) Extract traceroute information from Surveyor data sets. (Various ad-hoc tools were used.)

2. *pathconvert* – generate symbol mapping file from path file (optional).
3. Modify symbol mapping file to change symbol names or add location information (optional).
4. *pathconvert* – generate Felix Topology File from path file and symbol mapping file.
5. *topomap* – read Felix Topology File and generate GIF file with rendering of the network.
6. *xv* – to view the GIF file.

2.7 Clock drift correction tool chain

The clock drift correction tool chain (shown in Figure 14) removes the effects of clock drift from a pair of paths between two monitors in opposite directions. The chain consists of the following steps.

1. Extract x-y files from the Felix Database or from the Surveyor data (Figure 15).
2. *fixdrift* – run on pairs of paths (path and opposite direction path). Generates new x-y files.
3. Save x-y files back into Felix Database as a new dataset.

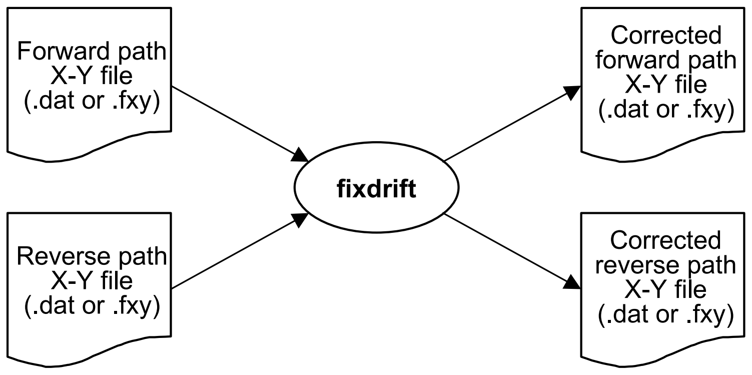

Forward path X-Y file (.dat or .fxy)
Reverse path X-Y file (.dat or .fxy)
fixdrift
Corrected forward path X-Y file (.dat or .fxy)
Corrected reverse path X-Y file (.dat or .fxy)

Figure 14. Clock drift correction tool chain.

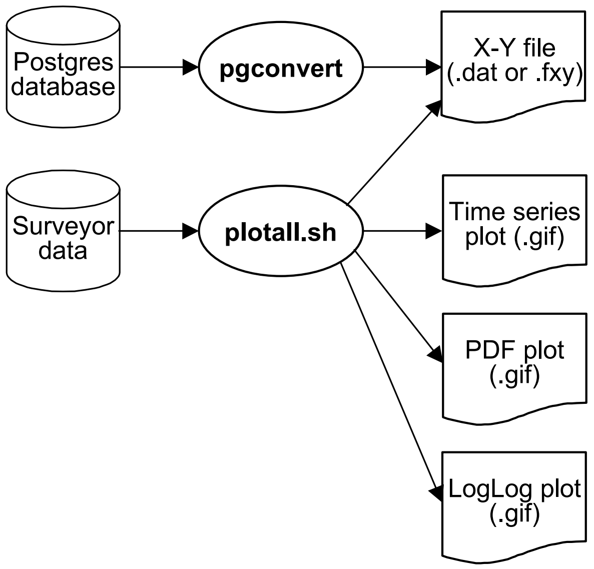


Figure 15. Extraction of Felix X-Y files

3. Felix network monitoring system

3.1 Monitor code and protocol

Features:

- Monitors exchange time stamped and sequence numbered UDP datagrams.
- Message sending policy is programmable among user-specified random interval or packetized voice traffic.
- Per packet results are archived in a centralized SQL database and optionally at the monitor itself in a flat file either with or without embedded HTML tags.
- MD5 authentication is performed over monitor to monitor messages.
- Monitors are capable of "learning" of the presence of other monitors and integrating them into the local view of the network.
- Auxiliary interface permits accessing the monitors via a web interface. Features of this interface include instructing the monitor to run traceroute, send timestamped email, or benchmark URL downloading characteristics. This interface will need to have a secure access protection added when run in an open environment

Discussion:

The monitor program uses UDP to receive and send network monitoring messages. The messages contain timestamps and sequence numbers in addition to other data, which allow the stations to track latency and packet loss of the sender and the other stations included in its data set. The program is both a sender and receiver. The receiver parses the incoming message and obtains records of the sending stations view of the network and updates both the local view of the network as well as updating the remote database. By analyzing the sending station's view of the network the receiving monitor can gain knowledge of previously unknown monitoring nodes and add them to the list of candidates to which it can send future datagrams.

The sender portion of the program has a choice of two sending policies. The packets are either transmitted based on a user programmable random interval or they can be sent based on a packet voice model. The voice model has several parameters among them are bit rate and number of voice channels simulated. With either policy the destination monitor is chosen randomly from the list of known nodes.

3.2 Archive server (including statistical data analysis engine)

Features:

- Centralized repository for monitor's observations.
- Responds to web based GUI queries by extracting the appropriate data from the SQL database and composing the requested graph type in browser ready format.
- Supports the following graph types: time series, probability density, autocorrelation, log-log distribution, and topology.
- Supports simultaneous requests.

The archive server (AS) is a concurrent server that processes database transactions placed by a web based GUI or the monitors themselves. The database is composed of entries generated by the monitoring stations. Each monitoring station periodically writes its local network view to the archive server; currently this is on a per packet basis. Requests are placed by users via a web based GUI which is implemented with an HTML form that in turn calls a CGI script. Queries can be for GIF Cartesian plots based on database contents, Cartesian plots of pre-existing files accessible to the AS, topology graphs, or delay and packet loss statistics in chart form. Plot types include time series of delay and packet loss, probability density, autocorrelation, variance time, and log-log distribution.

There are two archiving modes, in mode 0, each monitor has an archive and all monitors send their first-hand information to all other archives through the MDEP. In mode 1, each monitor sends its data directly to a designated archive server. There may be several regional archives. In the initial design, the LDA can only access data from one archive from which to construct the topology.

Interfaces:

User: The user interface is implemented as an HTML file called `felix.html`. Queries specify plot type, statistic type, beginning timestamp of the period of interest, end of interval timestamp, a pair of monitoring station host names, and the hostname of the archive server. The result of a plot type query is an HTML page that is composed on the fly and contains an XY plot in GIF format of the requested data. Chart type queries are expressed in a similar fashion but result in a text based chart containing Mean, Median, Variance, Standard Deviation, and Coefficient of variation. Topology graphs are expressed as ASCII files containing nodes and edges (network elements and links) and related information. The detailed format is given in the appendix below. The program that converts these files to .GIF format graphical objects for display by the GUI will be integrated. Requests are passed off to a child process which proceeds to parse the database management system (currently Postgres) for the appropriate data, forming a file composed of the dependent variable as it goes. The various mathematical functions are run on the y-axis values generating an (x,y) file. A command file is composed on the fly and passed to GNU plot along with the (x,y) file. This results in a GIF image, which is returned to the CGI script. The CGI script then composes an HTML page, in real time, returning to the browser the URL of the graph. The browser retrieves the graph via the URL and displays the web page containing the graphics. In the case of a chart the procedure is similar except the GNU plot step is replaced by a function that formats the table in HTML.

Process: The AS interacts with the GUI cgi program, the monitoring station processes, and the Topology Discovery process, as well as other archive server processes. The server is concurrent, spawning child processes to handle all requests. The GUI interaction is discussed above. The contents of the packets are in the same format as the packets that are exchanged between the monitors themselves. They are then written into the database file keyed by source-destination IP number pair (dotted decimal) and timestamp. Requests for topology are passed in a single character buffer in the same format as it is received in from the GUI to the LDA process via a socket interface. The results are returned to the AS and ultimately to the CGI script in a GIF format which embeds the images in an HTML page

3.3 Database

File Format: Felix Postgres Database

The dbm database management system (DBMS) was replaced by the PostgreSQL DBMS due to dbm's slow retrieval times and lack of query flexibility. PostgreSQL is accessed via the standard SQL92 language. It has a console type interface which is convenient as well as a Berkley socket interface for remote access by the Archive Server. The entries have the same field structure as the dbm database supported. The PostgreSQL DBMS manages the access to the data and in doing so removes the need for external semaphores synchronizing the reader and writer clients.

The SQL format for the data manipulation is also an advantage. Formerly, the data was organized based on a triple of source, destination address and time stamp. This is no longer the case and gives us significantly more flexibility in the way we deal with the data. Since the DBMS is handling the data manipulations there is no preformatted file type. The SQL interface allows us to retrieve data in any format that is convenient at the time. For instance, entire database entries could be retrieved or simply (x,y) files could be generated directly from the database based on the particular SQL query executed.

Postgres database file entry format:

<sender	- string>	dotted decimal
<receiver	- string>	dotted decimal
<timestamp	- string>	ctime() format
<end timestamp	- string>	ctime() format
<delay	- int>	mSec
<pkts. Lost	- int>	
<bandwidth	- int>	kb/sec
<pkts received	- int>	
<monitor version	- string>	

3.4 Web-GUI

Features:

- Statistical Graphs
- Control protocol for monitors (future)
- Topology discovery engines
- Network performance analysis tools

This *felix.cgi* script is used to respond to forms generated by `felix.html`. Upon reception of the query string from the browser HTML form the CGI script opens a socket to the appropriate archive server. A TCP packet is transmitted to the archive server which returns a fully processed binary file containing the requested graph in GIF format. This script then composes an HTML page on the fly and presents the GIF graph and its context to the browser for display. Included in the content is the query that resulted in the graph as well as the time and date of rendering. The graphs can be saved to disk with the normal browser functionality.

The web interface itself is implemented in HTML frames. The browser window is composed of three individual frames. The top frame is occupied by the Felix header and Telcordia logo and remains static. The left margin contains links to the various plotting engines and also remains unchanged during execution. The right side of the browser is occupied by whichever plotting engine is currently selected. The default is the Time Series plotting engine. The right frame is replaced by the results of the plot request upon submission of the form. In the HTML form, a time range and monitor pair are specified as well as the hostname of an archive server. The CGI program queries the archive server for the appropriate data and the browser displays the resulting graph along with the corresponding query and time of execution.

4. Monitor deployment, data collection and statistical analysis of data

4.1 Monitor deployment

Felix monitors were initially deployed within the Telcordia network to test our capabilities in a limited but realistic environment. Later, the deployment shifted to a collection of universities that were actively solicited with the goal of achieving a sufficient monitoring density to provide good coverage of the internet. This density was not actually achieved mostly due to inaccuracies in the clock synchronization required to perform the necessary one way delay calculations. It was at this time that a collaboration was formed between the Felix project and Advanced Network & Services, Inc. (*Advanced*). The benefit to the Felix project was the use of the *Advanced* dataset. The *Advanced* dataset is very valuable in that each of the 40 or so *Advanced* monitors contains a Global Positioning System (GPS). The GPS units have very precise clock synchronization which alleviates the difficulties associated with the one way delay measurements.

4.2 Data collection

For most of the problems, the Felix project is using input from two sources: data collected by Felix monitors; and data collected by the Surveyor Project, from Advanced Network & Services, Inc. Input data may also be generated by a simulator.

Our early work used the data from the Felix monitors, which were developed specifically for the Felix project. Felix monitors are distributed about the network to be monitored and mapped. The monitors send UDP packets to each other and collect delay and loss information. The delay and loss information is collected in a central location, called the archive server. Each data element consists of measurements for a single packet and is a tuple consisting of a source identifier, a destination identifier, a time stamp (the time the packet was sent), the measured delay between the source and destination, and the number of packets lost since the last packet received from the source by the destination. For timing, the Felix monitors use the local clocks on the Unix hosts on which they are running. These clocks are often not very accurate due to clock skew and clock drift. The clocks are either kept synchronized using the Network Time Protocol (NTP) or are not synchronized at all. In the NTP case, we have observed variations between clocks on different monitor hosts on the order of ±10 milliseconds, which is at the same order of magnitude as some of the measurements. Without NTP, we have observed differences of several minutes for the local clock times on different hosts. Because of these variations, we have done some investigations of methods to subtract the effects of clock drift from delay measurements. This is discussed in more detail in the next section.

For future research, we will use the Surveyor data, which is based on much more accurate timing: the Surveyor boxes contain GPS receivers. We recently signed a nondisclosure agreement with Advanced Network & Services, Inc. so that we could use the Surveyor data. *Advanced* claims that their delay measurements are accurate within 50 microseconds.

The input to the topology discovery problem is a list of time, delay and loss values for each source/destination pair.

The source and destination monitors are identified by their IP addresses, host names, or other aliases.

The time value is the clock time at which the current probe packet was sent by the source monitor (the sender) to the destination monitor (the receiver). This value is one of the fields in the probe packet. The receiver uses it to compute the delay – the transmission time – for the packet. The time value must be accurate so that the delay calculations will be accurate and so that measurements on different paths (that is, between different source/destination monitor pairs) can be correlated.

The delay value is the difference between the time that the probe packet was received and the time value included in the packet. The delay value is associated with the time value at which the packet was sent.

The loss value is the count of probe packets lost between the arrival of the previous probe packet received by the destination monitor from the same source as the current packet and the arrival of the current probe packet. The loss value is associated with the time value at which the current probe packet was sent. There are at least two ways to detect packet loss:

1. The packets sent by the Felix monitors contain sequence numbers. If two consecutive packets received at a destination monitor from the same source do not have consecutive sequence numbers, then the difference between the sequence numbers minus 1 is the number of packets lost. In the Felix monitoring system, packets are sent so infrequently that out-of-order arrival of packets is not an issue.
2. In the Surveyor system, the senders and receivers both keep track of when packets are supposed to be sent. If the receiver does not receive a packet within a certain time of its expected sending time, then it may be considered lost. (Actually, in the Surveyor project, they define such packets as having large/infinite delays and do not identify them as being lost.)

4.3 Statistics of network delay data

4.3.1 Local Dataset from Telcordia Network

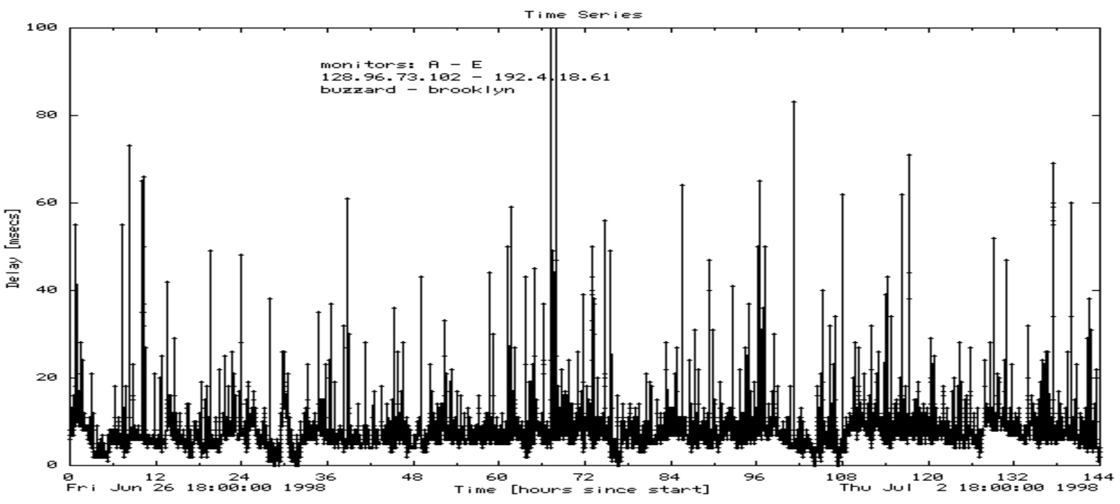


Figure 16 Typical trace of delay measurement (6 days, monitors separated by 50 miles)

Figure 16 shows a fairly typical trace of delay data taken between two Felix monitors located at separate Telcordia buildings in New Jersey. Interesting features of the data include the very spiky nature of congestion events. Even when the network is congested, the delay will sometimes be measured close to the minimum (i.e., propagation) delay between the monitors. The ragged bottom margin of the trace is evidence of clock drift between the two monitors' host computers. Clock drift has a relatively low-frequency effect, compared to packet delay in the network. Removing the clock drift effects is not as simple as low-pass filtering the data (though that method might be approximately correct), since it is possible that some low frequency effect is due to network delay as well as clock drift. The evidence that the lower margin wander is clock drift (as we will discuss in the next section) is the fact that we see symmetrical wander in the trace between the same two monitors in the opposite direction. In this trace, it is possible to see a component of cyclic modulation due to network load changing with the time of day.

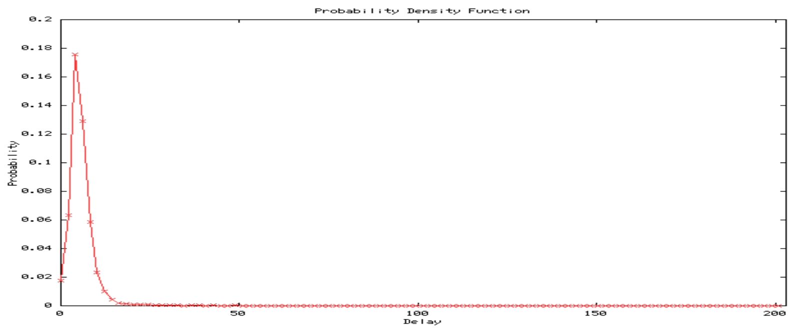


Figure 17 Probability density function (histogram) of delay data shown in Fig. 14.

4.3.2 Heavy-tailed distributions common in measured delay data

The probability density function (pdf) graph of this trace (Figure 17) shows a single mode around 10 msec. To see the characteristic decay of the tail of this distribution, we can plot log(1-F(delay)) vs. log(delay), the so-called "log-log" plot of the distribution, where F(d) is the cumulative distribution function of the delay. Figure 18 shows the typical behavior of data with heavy-tailed distribution. If the tail of the pdf decays exponentially, or as the exponent squared (as in a Gaussian distribution), then the log-log plot would show a rapid downward parabola. An ideal pdf curve with a logarithmic decay, would be typical of the Pareto distribution (i.e., the pdf behaves asymptotically as $f(x) \sim x^{-\alpha}$, rather than $f(x) \sim e^{-\alpha x}$). In the log-log graph the so-called "Pareto", or *heavy* tail behavior is identified as a linear slope. Empirical distributions that are heavy-tailed often end with a brief sharply declining tail. This graph is and exception to this rule because, even at the very end (where data points are sparse and inaccurate), the slope is quite horizontal. (More typical is the data shown in Figure 20.) In any case, the important portion of this distribution is the long section that appears approximately linear (with a slight bend down in the middle). In this region, from approximately (x = 0.8, y = -0.5) to (x=1.8, y = -3.0), the probability is diminished by a constant multiplicative factor for each constant factor in delay. In particular here, the probability drops by 2.5 orders of magnitude (a factor of 300) per decade (factor of 10) in the probability that the delay is greater than some particular value.

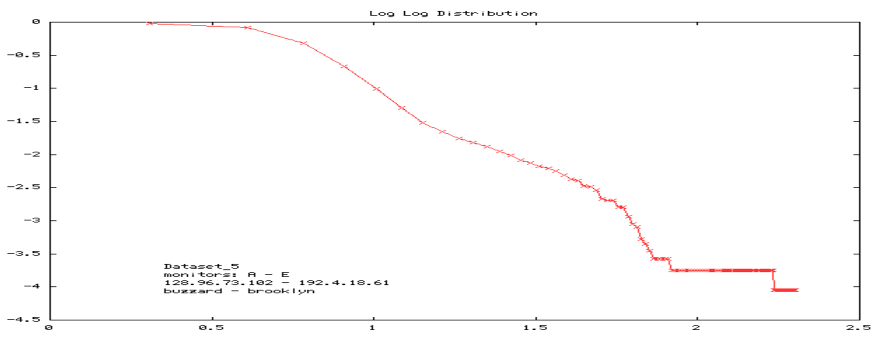


Figure 18 Log-log plot of delay distribution. Sections with approximately linear slope indicate heavy-tailed distribution of delay data.

Heavy tailed distributions, though apparently common in real-world data, are problematic in statistical analysis because the probability of apparent "outlying" events is much greater than it would be with typical Gaussian or exponentially distributed data. For example, this fact plays a role in our *cross-correlation* method for typology discovery. Consider a congestion event on a link in the network, and the effect it has on delay data measured between monitors. All of the monitor pairs whose path crosses the congested link will "see" the increased delay in their data. We could deduce that there is a congested link shared by these paths by correlating the data across pairs of paths and look for highly correlated periods of time. Now, if the delay data were Normally distributed, we would see a contribution to the correlation from all of the data points, both high and low delays, because when correlated, they move together. If the data is distributed with a heavy tail, the correlation calculation tends to weight the high-delay data points much more, simply because they are much farther from the average delay than in the Gaussian case. This makes the whole measurement less *robust* statistically, because it becomes very sensitive to the behavior of a very small number of high-delay data points.

Another interesting observation is that, although most delay traces show heavy-tailed distributions, some do not. Apparently when paths are quite lightly loaded, the delay distribution is (approximately) exponential or light-tailed.

4.3.3 Wide-area dataset from U.S.

The data we collected (not show) from monitors at universities across the United States typically showed extreme clock drift effects, to the extent they were unusable. An interesting area for further work (that we unfortunately did not have time to pursue) would be to evaluate whether the clock drift correction algorithms could produce useful data from these measurements.

4.3.4 Advanced "Surveyor Project" Data

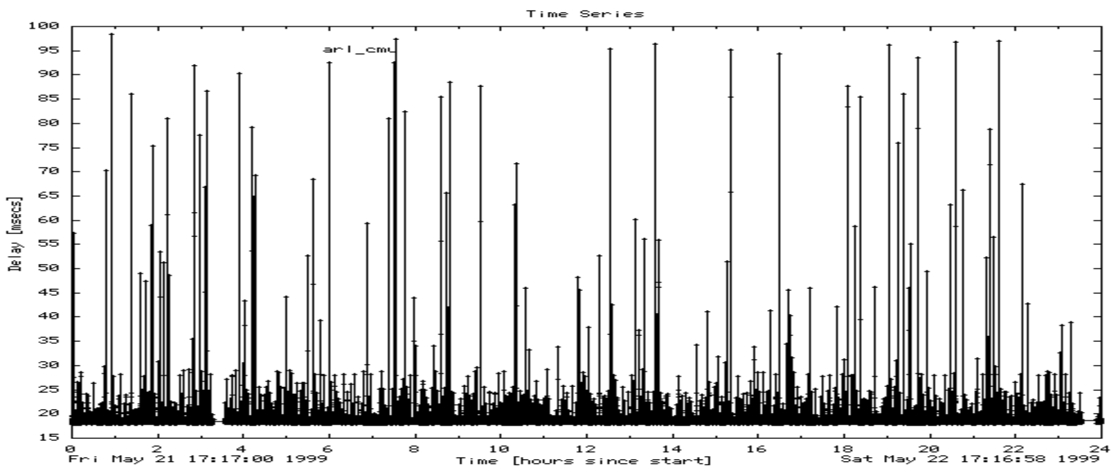


Figure 19 Typical delay trace from Surveyor monitors, courtesy of Advanced Network & Services, Inc.

The data obtained by agreement with Advanced Network & Services, Inc is illustrated in Figure 19. Note the flat lower margin, indicating no clock drift between monitors. The Surveyor monitors are synchronized using GPS receivers on each monitor. This trace shows 24 hours of delay measurements between an Army Research Lab site in Maryland and CMU in Pittsburgh Penn. Figure 20 show the log-log plot. Here the probability drops by only a factor of 13 for each decade decrease in delay, indicating a much stronger heavy tail than the data shown in Figure 16.

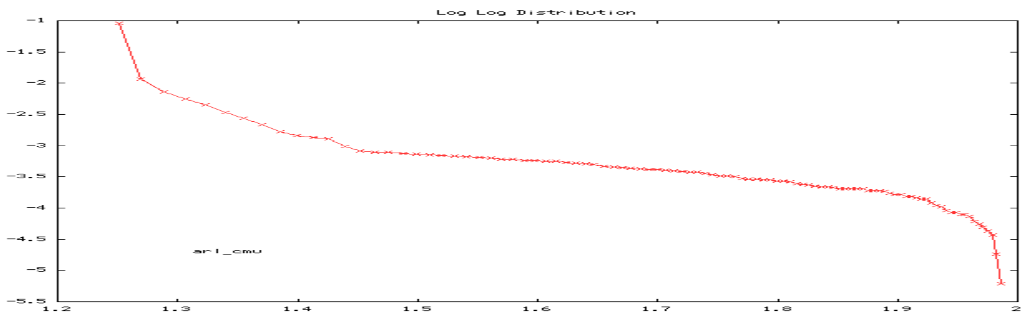
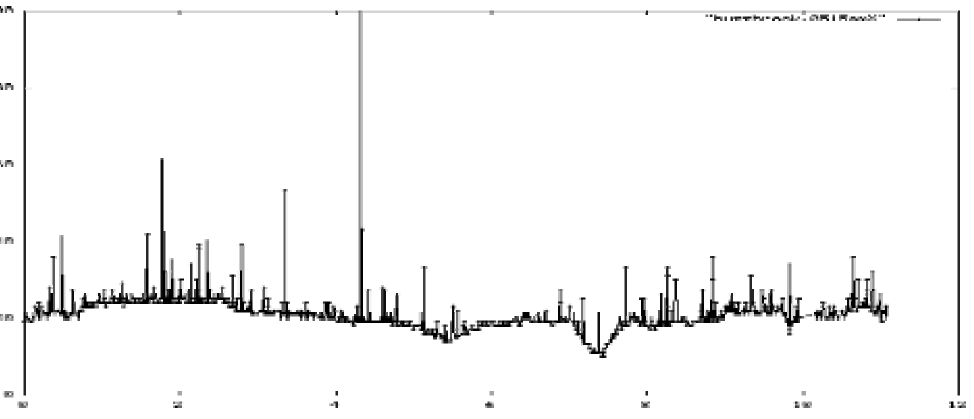


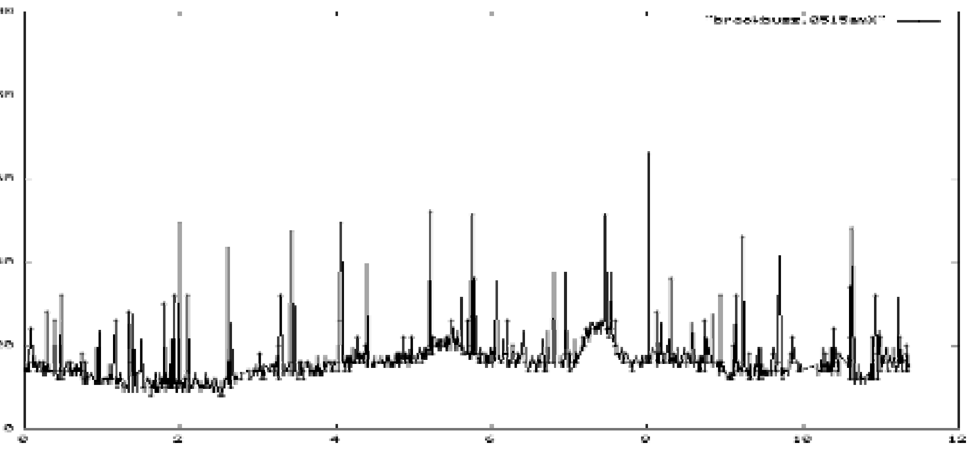
Figure 20 Log-log distribution of Surveyor data showing extremely long, heavy tail.

5. Clock drift and method for compensation (more lessons learned)

In our early analysis of data collected by the Felix monitors, we observed that clock drift had a significant effect on the delay measurements. There were two indicators of the clock drift problem. First, we expected that the time series plots of delays would have a flat bottom which would be the minimum time for a packet to traverse the network, i.e., the transit time when there was no congestion in any of the routers along the path. However, we observed that this was not the case (Figure 21a or Figure 21b); the bottom of the time series curves (which we shall call the lower envelope of the curve) drifted up and down. Second, we observed that the lower envelopes of time series plots for delays between the same pair of monitors in opposite directions had the same shape but with one of the directions inverted (Figure 21). Below is some analysis showing that this is indicative of clock drift. We will also describe an algorithm for removing a large portion of the clock drift.



a) Delays from buzzard to brooklyn, 15 May 1998, midnight until noon.



b) Delays from brooklyn to buzzard, 15 May 1998, midnight until noon.

Figure 21. Evidence of clock drift. The lower envelope of the delay time series in one direction of transmission displays a strong component that is exactly inverted in the other direction.

5.1 Analysis of inverted patterns for measurements in opposite directions

Consider measurements between monitors A and B.

Given:

When there is no congestion in the network, the network delay from A to B is X..

When there is no congestion in the network, the network delay from B to A is Y.

	Actual time	Time measured at A	Time measured at B
Packet sent from A to B	t	t_A	t_B
Packet received at B	$t + X$	$t_A + X$	$t_B + X$

Table 1. Timing for packet sent from A to B on an uncongested network with synchronized clocks.

	Actual time	Time measured at A	Time measured at B
Packet sent from B to A	t	t_A	t_B
Packet received at A	$t + Y$	$t_A + Y$	$t_B + Y$

Table 2. Timing for packet sent from B to A on an uncongested network with synchronized clocks.

	Actual time	Time measured at A	Time measured at B
Packet sent from A to B	t'	$t_A + (t' - t)$	$t_B + (t' - t) + D$
Packet received at B	$t' + X$	$t_A + (t' - t) + X$	$t_B + (t' - t) + D + X$

Table 3. Timing for packet sent from A to B on an uncongested network. Clock at B has drifted forward.

	Actual time	Time measured at A	Time measured at B
Packet sent from B to A	t'	$t_A + (t' - t)$	$t_B + (t' - t) + D$
Packet received at A	$t' + Y$	$t_A + (t' - t) + Y$	$t_B + (t' - t) + D + Y$

Table 4. Timing for packet sent from B to A on an uncongested network. Clock at B has drifted forward.

Initially, we assume that the clocks at A and B are synchronized at time t, i.e., $t_A = t_B = t$. Thus for a packet sent from A to B at time t with an uncongested network (Table 1), the measured delay is

$$(t_B + X) - t_A = (t_A + X) - t_A = X\ .$$

Similarly, for a packet sent from B to A (Table 2),

$$(t_A + Y) - t_B = (t_B + Y) - t_B = Y$$

Next, assume that, at time t', later than time t, the clock at B has drifted forward D units, while the clock at A has stayed synchronized with the actual time. For a packet send from A to B at time t' with an uncongested network (Table 3), the measured delay is

$$(t_B + (t'-t) + D + X) - (t_A + (t'-t)) = t_B - t_A + D + X = \underline{X + D}\,.$$

Similarly, for a packet sent from B to A (Table 4),

$$(t_A + (t'-t) + Y) - (t_B + (t'-t) + D) = t_A - t_B + Y - D = \underline{Y - D}$$

Thus, adding a clock drift at monitor B increases the delays observed at B and decreases the delays observed at A by the same amount. Similarly, if the clock drift at B were negative, the delays observed at B would decrease and the delays observed at A would increase by the same amount. This could be generalized for combinations of drifts at both A and B. This explains the inverted lower envelopes for the time series plots for delay measurements between pairs monitors in opposite directions.

5.2 Algorithm for removing clock drift from delay measurements

Given delay measurements between two monitors in both directions (Figure 22), it is possible to remove a large portion of the clock drift using the following algorithm:

1. For the data from each direction, break time up into fixed length intervals such that each interval typically contains several measurements (enough that one of the measurements in each interval is likely to occur when there is no congestion between the monitors). The starting times of the intervals for both directions should be aligned.

2. Determine the minimum delay measurement in each interval (Figure 23). For intervals containing no data, the average of the minimum measurements for the surrounding intervals (the nearest intervals that do contain data) can be used. This step filters out the high frequency changes and finds the lower envelope for the time series curve.

3. For each direction, compute the mean (or median) of the minimum delay measurements of all the intervals.

4. For each direction, subtract the mean (or median) from the minimum delay measurement in each interval. This gives an approximation of the clock drift observed at the receiving hosts (Figure 24). (This assumes that the minimum delay between the hosts is being observed for at least one of the measurements in each interval.) The approximations of the clock drift observed in each direction will have opposite signs and will not be exactly the same.

5. For consistency, we wish to adjust the measurements in both directions by the same amount. Thus we compute an “average” of the two clock drift approximations: Subtract the drift approximation for the second direction measurements from that of the first direction

measurements (since they have opposite signs) and divide by two (Figure 25). (There may be better ways to combine the two approximations.)

6. For the first direction, subtract the "averaged" drift approximation from the original data for that direction (Figure 26). Since there is only one drift approximation for each interval, the same drift approximation is used for all the original data points that fall within an interval.
7. For the second direction, add the "averaged" drift approximation to the original data for that direction (Figure 26).

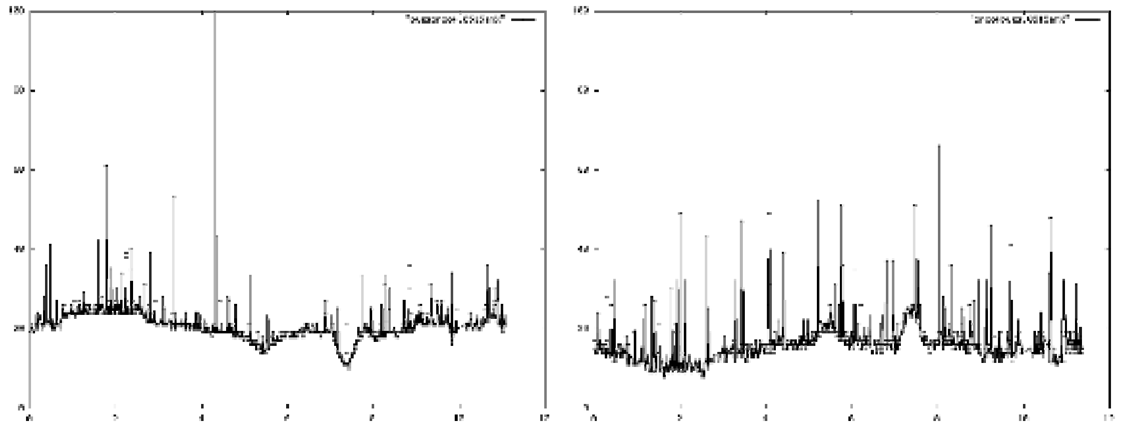


Figure 22. Original delay measurements.

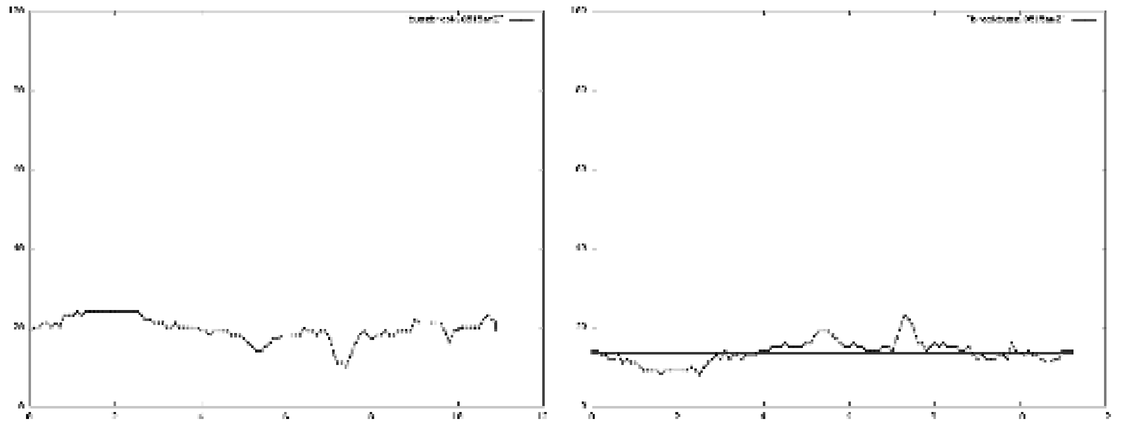


Figure 23. Lower envelopes of delay measurements.

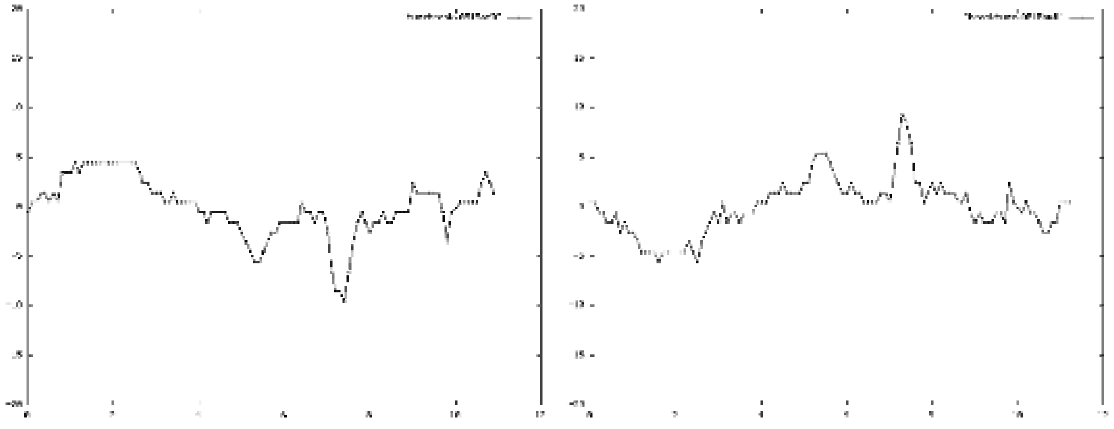


Figure 24. Clock drift estimates in each direction.

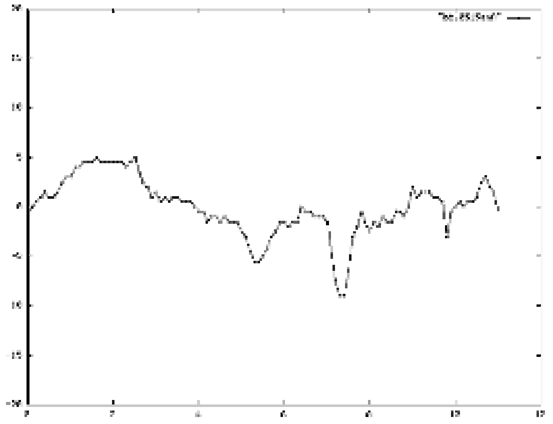


Figure 25. Combined clock drift estimates.

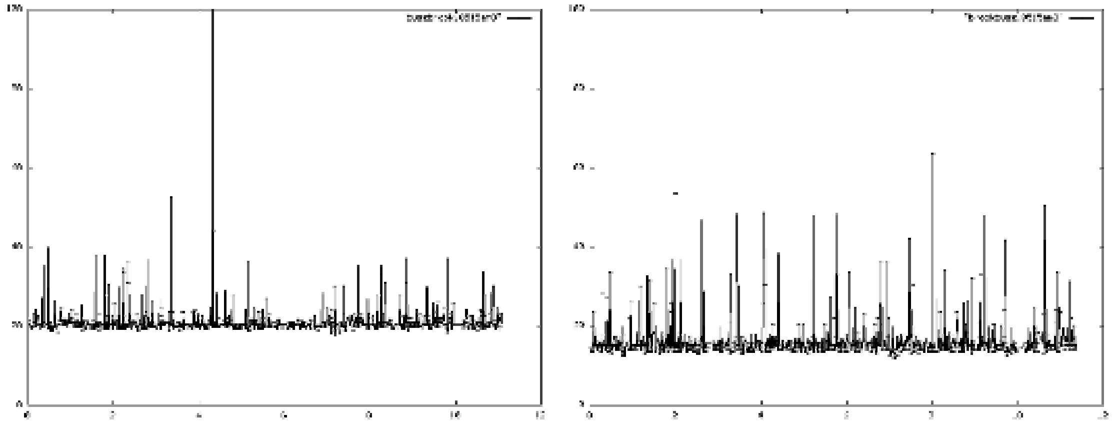


Figure 26. Delay measurements after removal of clock drift.

6. Topology discovery and performance assessment algorithms

The goal of the **topology discovery** problem is to convert delay and/or loss measurements into a net list that identifies the monitors and intermediate nodes and the links that connect them. We have worked on several approaches for solving or partially solving this problem. In the partial solutions, we decompose the problem into two or more sub-problems and try to identify solutions for the sub-problems (see Figure 27). Solving sub-problems produces intermediate results:

- The *common component matrix* identifies source/destination paths that have common components. The axes of the matrix are the network paths, and the values are 0 for no components in common or 1 for components in common.
- The *path component matrix* identifies the components (the links) that make up each source/destination path, but does not identify the order of the components. The axes of the matrix are the paths and the links of the network. Values of elements are again 0 or 1.

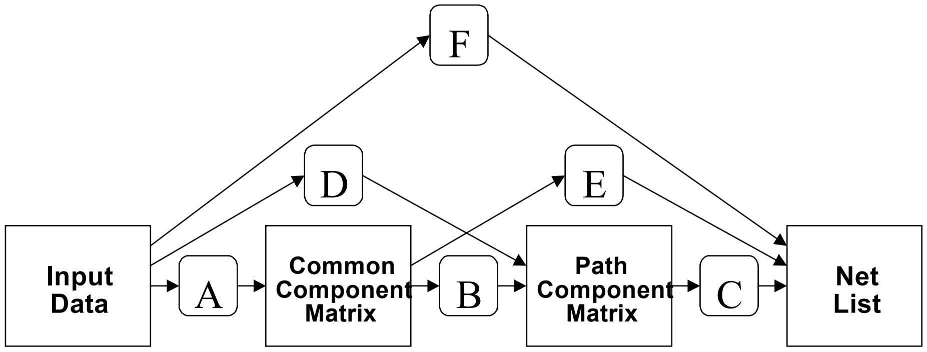


Figure 27. Decomposing the Topology Discovery problem. Complete solutions from delay data to network graph are given by combinations: F, DC, AE, ABC.

The **performance assessment** problem is to quantify the "health" in some sense of the network in terms of each link. Some candidate metrics for link performance are *delay*, *loss*, *load* and *throughput*.

The six algorithms or *methods* we have devised address topology discovery and performance assessment as follows:

Matrix method	Topology discovery, sub-problem D
Tree-growing method	Topology discovery, sub-problem F
Spike-tail method	Performance assessment of link load
Correlation method	Topology discovery, sub-problem D
Matroid method	Topology discovery, sub-problem C
Distance-realization method	Topology discovery, sub-problem F

6.1 "Matrix" or Linear Decomposition Method

Linear decomposition finds a solution to $\mathbf{A} \times \mathbf{x} = \mathbf{b} + \boldsymbol{\varepsilon}$ that minimizes error (vector $\boldsymbol{\varepsilon}$), where $\mathbf{A}$ is the path component matrix, $\mathbf{b}$ is a vector containing the delay measurements for each of the source/destination paths, and $\mathbf{x}$ is a vector of the delays contributed by each of the links making up the paths. In the overall problem, the $\mathbf{b}$ vector is the empirical data. Solving for $\mathbf{A}$ is sub-problem D of topology discovery, and solving for $\mathbf{x}$ is the performance assessment problem in terms of link delay.

6.1.1 Fundamental under-determination of link delays based on path delays

Given $\mathbf{A}$ and $\mathbf{b}$, we have two approaches for solving for $\mathbf{x}$: one is a direct solution through Gaussian elimination or matrix factorization techniques, and the other is a relaxation technique. We can only find a unique solution **x** if **A** has rank (number of independent columns) equal to the dimension of **x**. Otherwise there will be non-zero vectors **y** such that **Ay=0**, and so if x is a solution with error $\in$,then x+y will also be a solution with the same error $\in$. Adding the constraint that the link delays should be nonnegative, we in general find that the set of solutions **x** to minimize $||\mathbf{Ax-b}||$, $\mathbf{x}\geq\mathbf{0}$ will be a polyhedron.

There are a number of possible sources of rank deficiency for **A**. If there are not as many paths as there are link delays being estimated, **A** will be rank deficient. If there are two links **e** and **f**, so that the set of paths using **e** is the same as the set of paths using **f**, then we will not know how to divide the delay between **e** and **f,** and **A** will be rank deficient. These rank deficiencies might be corrected by placing more monitors. Possibly the most important rank deficiency arises when, as in our usual model, we allow the delay along a link in one direction to be independent of the delay along that link in the other direction, and we have any nodes in the network besides monitor nodes. Of course, in any non-trivial internet there are internal nodes (routers) and the delays in each direction of a link are independent because the traffic creating congestion delays in each direction is independent of the other. In this situation, at any internal (non-monitor) node we can add a constant **c** to the delay of every link entering that node, subtract **c** from the delay of every link leaving that node, and not change path delays. The allowable values for these constants are limited by the requirement that link delays be nonnegative. There may be other sources of rank deficiency.

6.1.2 Simulated Annealing

Solving for both $\mathbf{A}$ and $\mathbf{x}$ is unlikely to find a unique solution if the values of $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{x}$ are fixed. However, if we take advantage of the fact that the values change over time, we may be able to make an initial guess for A and refine it as new information is added. We call this approach the *annealing* method.

This approach makes an initial guess of the components making up each of the paths, constructing a path component matrix, A. Then it attempts to find a solution to the problem **Ax** = **b** + ε that minimizes error, ε, where x is a vector containing the delays for each of the components, and b is a vector containing the end-to-end delay measurements. Annealing is used to improve the guess for the A matrix (and the **x** values). The result of this approach is a path component matrix. Additional steps are required to determine the connectivity (e.g., the order of connection) of the components.

We have not pursued this approach beyond identifying it as a potential methodology.

6.2 Tree growing method

The tree growing approach is an algorithm to transform the input data directly into a net list, with the limitation that the graph is a tree. While there will in general be man y topologies that can have link delays assigned to them to exactly fit given total path delays, if there is a tree that can fit this data, it will be the unique tree that can, and it will be easy to find this tree.

The tree growing method assumes that the network is a tree, but that there may be some error in the measurements of path delay, including varying queuing delay. The approach attempts to build a graph (tree) that fits the delay measurements by adding one monitor and link at a time, inserting the other end of the link somewhere in the existing tree. The insertion point is chosen to minimize the sum (over individual packets) of squared errors, allowing link delays to vary. One can require that link delays be nonnegative, at the cost of the additional time to solve nonnegative least squares problems rather than unconstrained least squares problems. The result of the method is a description of the connectivity of the monitors and the intermediate nodes.

One question not fully resolved is when a new monitor and link should be attached to a node already existing in the tree, rather than to a new node subdividing an existing link. The squared error will always be less if we subdivide an existing link, giving a tree where all the internal nodes have degree three. If one of the two parts of the subdivided link has a very small delay, in the actual network the monitor and link was probably connected directly to the preexisting node on that end of the subdivided link. The current implementation allows for a threshold below which links are contracted; any link of total delay less than this threshold may be removed, and its endpoints made into a single node. It appears that an actual statistical test could be performed here, if the error distribution were known.

The tree growing approach finds a unique optimal topology when the data actually corresponds to a tree, and has small error. To keep the error small, the algorithm should probably be fed minimum delays rather than average delays. The least-squares techniques used will suffer if average delays including heavy-tailed queuing delays are used, since least-squares implicitly assumes normal error. If average link delays are required, they should be fit to a topology determined from minimum delays. Because of the rank deficiency of the path-component matrix, the directed link delays produced will only be one point in a polyhedron of link delay vectors that fit the data equally well.

This method was implemented as the program *fxtree*, described in the Appendix.

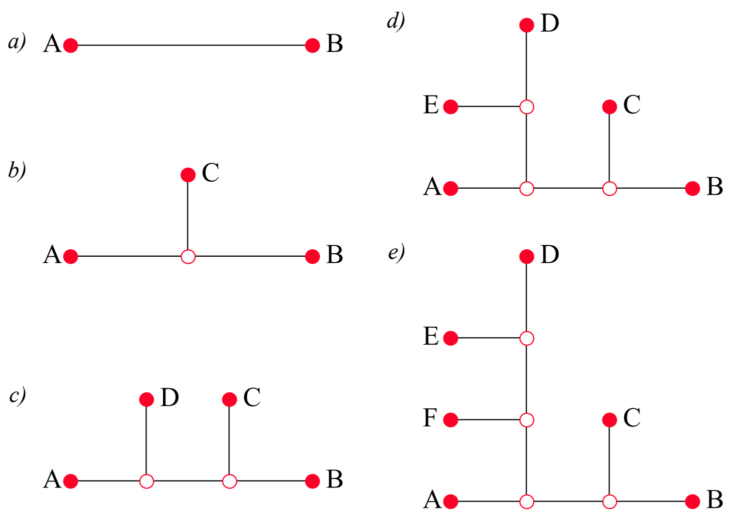


Figure 28. Successive steps in Tree Growing approach to topology discovery.

6.3 Spike-tail method

In this method we try to infer link loads from the observed sample distributions of packet delays on the various monitor-to-monitor paths. It is assumed that the network topology and routings are known, and we are interested in possible congestion in the network.

We expect the sample distribution of packet delays on a path to consist of a spike (e.g., as in Figure 17), corresponding to packets experiencing identical propagation delays but no queuing delay, followed by a continuous tail, corresponding to packets experiencing additional queuing delay. The expected value of the fraction of packets in the spike will be the probability that the packets saw no queuing delay.

In general, the load on a link expressed as a fraction of link capacity is equal to the probability that any given packet crossing the link will have to wait for the link to become free, that is, will have queuing delay. The probability that a packet along a path sees no queuing delay should be the product of the probabilities that it sees no queuing delay on the individual links. For this method we assume that, given the link loads, the delays seen by each packet at each link are independent.

In this method we estimate this link load by:

1) determining which packets see queuing delay;

2) formulating a likelihood maximization problem, to choose the individual link queuing delay probabilities to maximize the likelihood of the observed path queuing delay frequency; and

3) solving this likelihood maximization problem using a general purpose nonlinear optimizer.

To further describe step 2, let $\mathbf{d_{i,P}}$ be 1 if packet **i** on path **P** is delayed, 0 otherwise. If the probability of a packet crossing link **e** being delayed is $\mathbf{p_e}$,, then the probability of a packet crossing that link having no delay is $\mathbf{q_e = 1- p_e}$, and the probability of a packet crossing path **P** without delay is $\mathbf{\Pi_{e\ in\ P}\ q_e}$. The probability of observing a vector **d** is

$$\Pr(\mathbf{d}) = \begin{cases} \Pi_{\text{all paths } P}\, \Pi_{\text{packets } i \text{ on } P}\, (\Pi_{e \text{ in } P}\, q_e) & \text{if } d_{i,P} = 0, \\ \Pi_{\text{all paths } P}\, \Pi_{\text{packets } i \text{ on } P}\, (1 - \Pi_{e \text{ in } P}\, q_e) & \text{otherwise.} \end{cases}$$

The maximum likelihood estimator for the probabilities **q** is that choice of values that maximizes this probability for the actually observed values of **d**.

There does not appear to be any analytical solution for this problem, so we turn to numerical solution. We performed some very preliminary tests using the general nonlinear optimizer MINOS, which found multiple solutions, depending on the starting point. All of these solutions gave the same likelihood value, suggesting that they were true multiple optima.

Issues still to be resolved include how to identify which packets are in the spike of the distribution in presence of noisy data, and the tractability of the likelihood maximization problem. In the present form, this problem is unsolvable, for the same reason that the Matrix method fails, that is the under-determination of the path-component matrix. The analogous demonstration of insufficient rank here is as follows: For any interior node, we can multiply the probabilities of no delay on incoming arcs by a small constant, divide the probabilities of no delay on outgoing arcs by that constant, and get the same likelihood of our observation.

We have not written code to implement this method because of these unresolved theory issues. The core useful idea here is the connection between the spike-tail weight ratio, which is observable for path delays, and the load on the links, which is a desirable measurement of performance. The problem is that we cannot uniquely decompose the path spike-tail ratio into equivalent quantities for the set of individual links.

6.4 Cross-correlation method

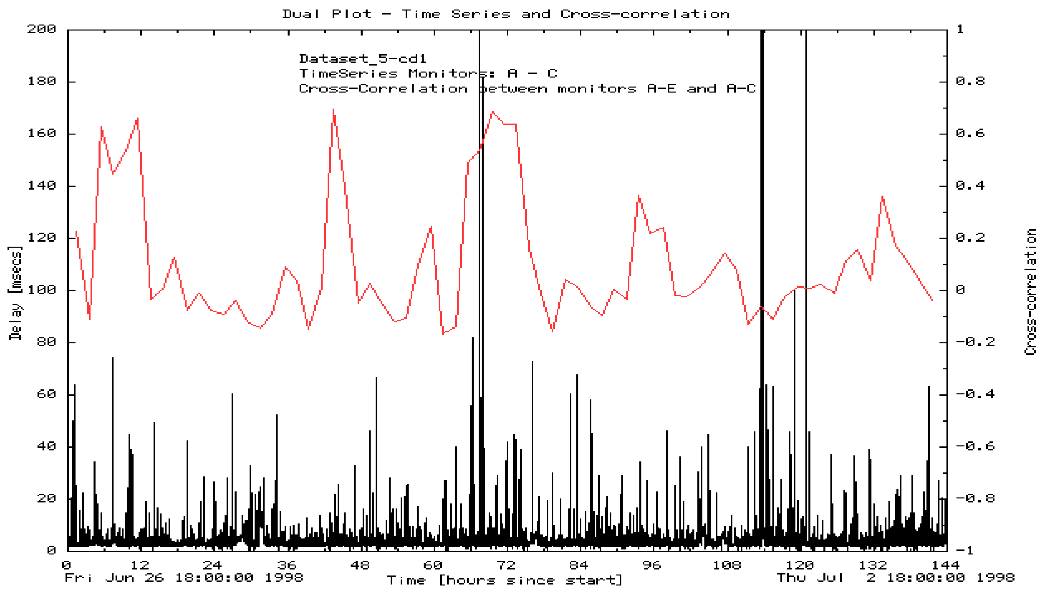


Figure 29 Cross-correlation function (top curve) showing periods of high correlation between two time series (one shown, one not shown)

6.4.1 General description of method

In this method for topology discovery, we investigate how we can take advantage of the time-dependent behavior of path delay, as a much richer source of observable information about the network than a simple scalar metric for each path (such as average or minimum delay). The basic premise is that congestion occurs in specific, identifiable *events* on each link that can be observed on all paths crossing that link, thus identifying the relationship between links and paths.

Consider the delay $d_{AB}(t)$ on the path from monitor A to monitor B as a function of time. This is essentially a signal with spectral or correlation properties, as well as stationary statistical properties. Now consider a windowed co-variance, or *cross-correlation* function between two delay time series on paths AB and CD,

$$r_{AB-CD}(\tau) = \frac{1}{\sigma_{AB}\sigma_{CD}T}\int_{\tau}^{\tau+T}\left[d_{AB}(t)-\mu_{AB}\right]\left[d_{CD}(t)-\mu_{CD}\right]dt$$

where μ_{AB} and σ_{AB} are the mean and standard deviation of the delay on path AB, and T is the duration of a sliding window used to capture the correlation around a specific point in time. Figure 29 shows a time series trace of delay measured in our network, and the cross-correlation function computed between this time series and another one (not shown). Note in this example the strong congestion event in the center of the trace, and another toward the end. In the first case, the cross-correlation function goes high, suggesting that the congestion is on a link that the two paths have in common. In the second case the cross-correlation stays low, indicating that the congestion is only on the one path whose time series is shown.

Hypothesis: when there is significant correlation between two paths, they share common elements (one or more links) and one of those links has become congested. When the common element is not congested, its existence may not be apparent in the correlation function because the common delay component is insignificant compared with other path delays. Therefore some level of network congestion is necessary for this method to work. In the absence of congestion on some links, a network topology may be discovered that omits these lightly-loaded links. Though not complete, such a result may still be useful. Each peak in the r_{xy} function then identifies a link common to paths x and y, though not uniquely. We can eliminate repeated occurrences of the same link by noting peaks that occur in the same combination of paths. If every link in the network congests at least once (and we can solve the problem of different congestion events coinciding in time), then we can identify all of the links, and which paths are associated with them. This is the path-component matrix, our intermediate result, from which we can devise another algorithm to produce a graph (for example, the Matroid method, described below).

6.4.2 Properties and challenges of correlation method

The combinatorics of this method are a significant issue. For $N_m = 7$ monitors, there are $N_p = N_m(N_m-1) = 42$ paths, and $N_p(N_p-1)/2 = 861$ path-pairs or instances of cross-correlation functions to evaluate (potentially). We clearly need to look for algorithms that structure the space of path pairs and avoid correlating every combination.

An initial experiment for the cross-correlation method is to see if we can easily distinguish disjoint paths from non-disjoint paths (i.e., sets of paths that share at least one common link). The average cross-correlation, i.e., r_{xy} taken over the whole trace without windowing, is a simple measure, and in our experiments, it was often close to zero for disjoint paths and higher where paths shared common links. We did find counter-examples though, where r is about zero despite common elements.

Another important issue for the correlation method is the effect of the **heavy-tailed statistical distribution** of network delay. Since we have seen empirical evidence of heavy-tailed distributions, we will need to proceed carefully in our utilization of the cross-correlation function. Cross-correlation is not “robust” in the sense that its use in statistics is usually predicated upon the assumption of Normally distributed data. When data has a heavy-tailed distribution, the effect is that data points far from the mean (which occur with higher probability than with a light-tailed distribution) will dominate the integral. The results are then essentially determined by a very small number of high-delay outlying events. The discovered topology may then lose some of its structure, showing only the most heavily congested links. Here are several ideas for compensating the data in order to improve the statistical robustness:

1. Model the delay distribution, and map the data back to a better-behaved distribution, such as Normal or exponential;
2. truncate distribution, treating very high values of delays as statistical anomalies;
3. divide the distribution into regions, and analyse the network behavior differently within each, then combine results;
4. use rank statistics or another non-parametric method;
5. use chi-squared test instead of least squares where applicable;
6. transform data using log, 1/x, square root, “cox-box” method, etc.

This investigation is left for future work.

For the cross-correlation method, the object is always to discover the path component matrix, i.e., to identify which links are on each path between two monitors. *We assume that the data corresponds to a **reduced graph***, as defined in Section 1.2.1 above. For a reduced graph, we have the general property that each link belongs to a unique set of paths. If a link congests alone, while the rest of the network is quiet, then the paths that cross that link will identify the congestion event. This property is illustrated in Figure 30, where a cross-correlation between several pairs of paths is shown. If we measure the correlation at the time of the congestion event between pairs of paths that cross the congested link, then the paths crossing the link will be identified. Of course, we may need to consider all pairs of paths to find all of the paths crossing that link. A simpler, interpretation of Figure 30 is to consider that each trace is simply delay, not correlation, showing congestion in common among various combinations of paths at different times. If we identify each unique group of paths (and we assume that links do not congest together), then each unique group corresponds to a single link, and shows when that link is congested. If we consider a long enough experiment that all links, in fact, do congest, then we will find them all, and we will know their relationship to the paths. This discussion illustrates (somewhat ideally) a concept that will recur throughout our investigation of the cross-correlation method.

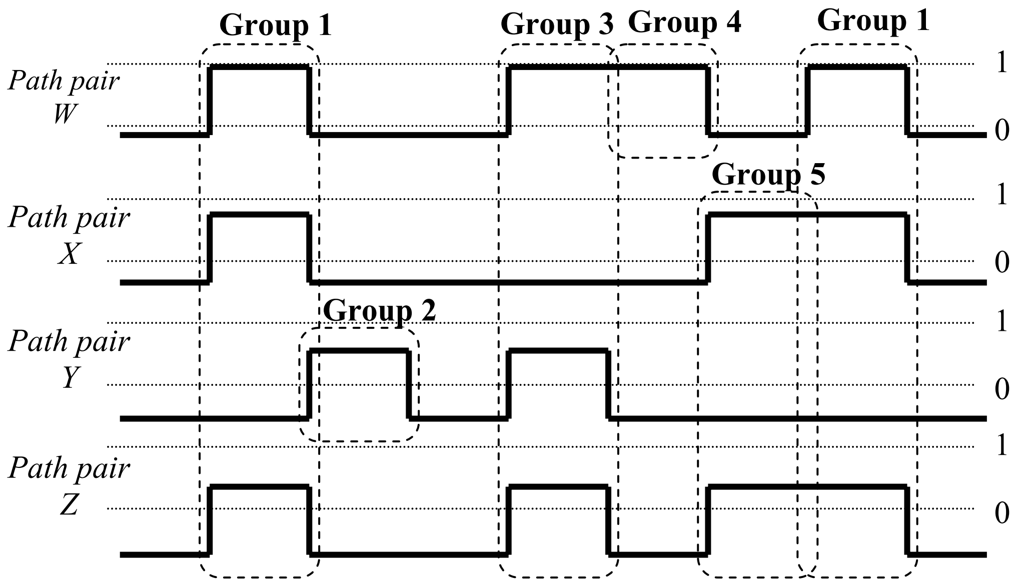


Figure 30 Grouping of congestion events across paths (observations) in cross-correlation method.

6.4.3 Break problem into progressive pieces

Our approach to developing general topology discovery within the Cross-Correlation Method (CCM) framework is to create a sequence of well-defined problems. We start with simple assumptions about network behavior that can be modeled in a simulation, and then build up to much more sophisticated algorithms that will work on empirical network data. Note, at this point, we have results on the first two cases. The others are sketched here in terms of modeling assumptions. Further investigation is left for future work.

CCM case 1, non-overlapping events. Assumptions:

1. Congestion events are strong and identifiable because the delay time series has a distinct “shape” as a function of time).
2. Congestion events occur on only one link at a time.
3. All links see congestion at some level.
4. Time is continuous, events occur at random with random duration.

CCM case 2, overlapping events in discrete time. Assumptions:

1. Time is counted in fixed-duration, discrete intervals
2. Links are either congested or not with independent, identically distributed probability in each time interval
3. Multiple links can congest at the same time (due to independence)

4. Link congestion probability is bounded above and below. (The range of these bounds, with successful topology discovery becomes a performance metric for the algorithm.)

CCM case 3, overlapping events in continuous time. Assumptions:

1. Congestion events are strong and identifiable because the delay time series has a distinct "shape" as a function of time).
2. Time is continuous, events occur at random with random duration.
3. Congestion events have well-defined starting and ending times.
4. Congestion events occur independently on each link, may coincide or partially overlap.

CCM case 4, congestion events with ragged edges. Assumptions:

1. Congestion events are strong and identifiable because the delay time series has a distinct "shape" as a function of time).
2. Time is continuous, events occur at random with random duration.
3. Congestion events occur independently on each link, may coincide or partially overlap.
4. Congestion events start and end more gradually. A threshold-crossing test will show event turning on and off several times before settling down. Start and end times measured on different paths for the same event will vary somewhat.

CCM case 5, use of realistic traffic models and cross-correlation function. Assumptions:

1. Traffic models are used (in a simulation) with heavy-tailed distribution of intensity (and therefore of delay imposed on Felix probe traffic), with long-range dependent time correlation.
2. Congestion events have properties (to be evaluated) consistent with the traffic models.
3. Cross-correlation function is used to show that congestion events measured on two paths are related.

CCM case 6, experiments with empirical data. Assumptions:

1. Measured Felix-style one-way delay data across a reasonably-sized network is the bases of improvements to the topology discovery methodology.

6.4.4 Solution to case 1, non-overlapping events

A simple brute force solution to this problem is to find each set of events that happen at the same time on different paths, and note what combinations of paths occur together. Then, assuming that all links are congested at least once, each unique combination of paths corresponds to a different link.

There may be issues in comparing events on different paths, especially if the location of an event in time is determined by thresholding the delay value. The most convenient representation for event observations on a path, when there is a very large set of data, may be to record the threshold crossing times. The event times on different paths will then differ slightly, since the probe packets are not synchronized.

The following algorithm is more elaborate, but has the minimum necessary complexity of *O*(number of paths × number of links × avg. number of events per path). This algorithm classifies events, assign them to paths and *groups* of links, that are eventually decomposed into single links. For illustration, Figure 31 shows a small example with two paths that share two links in the middle of the network.

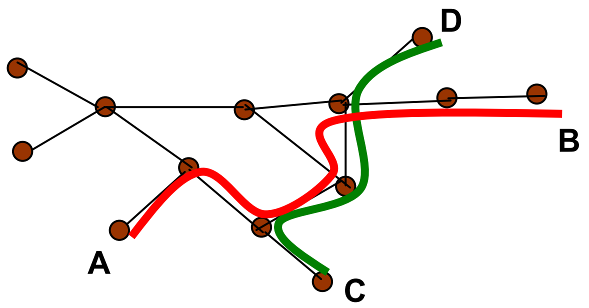


Figure 31 Network illustrating intersecting paths that share two links.

Algorithm. A network graph is composed of monitors (end points), links, (internal) nodes. A path is the ordered set of links that a packet traverses from a source monitor to a destination monitor.

Congestion "events" occur on each link and are measured by observing the packet delay along a path. The events belonging a path are the union of events occurring on each link on the path. Assume events are discrete, well-defined in time, and non-overlapping. Two events on different paths can be identified as the same event if they coincide in time.

A "group" is a set of events that are caused by some contiguous set of links. A link is *associated* with a group if its events are in the group. A path is *associated* with a group if it contains at least one link that is associated with the group. The events associated with a path constitute a group.

Initially no links are considered "discovered". The algorithm will eventually discover each link, the paths that cross it, and the events associated with it.

```
Repeat for each path, $P_i$, for i=1 to number of paths, n:
        Form a group, $G_k$ consisting of the events on path $P_i$. Delete any events in common with
        links previously "discovered".
        $P_i$ is associated with group $G_k$.
        Begin routine to break up the group into smaller groups as follows:
        Breakup(group $G_k$, path $P_i$):
                For each subsequent path, $P_j$, for j = i+1 to n
                    Compare events of group $G_k$ to events of path $P_j$
                    If any common events are found, then {
                        Delete the common events from group $G_k$
                        If no events are left, then $G_k$ is a null group.
                        Form a new group, $G_{k+1}$ containing only the common events
                        $G_{k+1}$ inherits association of all paths associated with group $G_k$
                        Path $P_j$ is associated with group $G_k$.
                                Break up the new group, execute routine Breakup($G_{k+1}$, $P_j$)
                                If $G_k$ is null then breakup is finished for $G_k$
                        }
```

After comparing G_k with the last path P_n, if G_k never became null, then G_k is the set of events from a single Link, and the set of Paths associated with G_k are the paths traversing the Link.

Continue for each new path P_i until finished.

End of algorithm.

In this algorithm, we start with a group of events corresponding to each path, and eventually break them down into smaller groups, one for each link. Each group that ends up being a link is compared to each path only once, so the complexity is proportional to the number of paths times number of links. (We could probably check this more rigorously…) Each discovered link must be compared to every path at some point to see if that path traverses the link. Since we do not assume any knowledge about the graph (that might restrict paths and links), we cannot assume that a path does not traverse a link (or a group along the way) unless we test it. The sequence of comparisons in this algorithm can be drawn as a tree (not shown) with the terminating leaves being links. Some of the intermediate nodes are links as well.

6.4.5 Solution to case 2, overlapping events in discrete time

This algorithm is much more complex than the previous case, since events can coincide in time. Some of the events observed simultaneously on a set of paths will identify single links, but sometimes the events will be due to simultaneous congestion on multiple links. The key to this algorithm is to discriminate statistically between the behavior of the true links and the false or *apparent* links that are really observations of several true links that happen to congest together.

We evaluate the algorithm for this case in a space defined by the following parameters:

1. Size of network discovered (number of monitors, links or paths)
2. Duration of delay time series needed (i.e., length of time simulated)
3. CPU time to compute topology
4. Memory required to compute topology
5. Upper and lower bounds on link congestion rate
6. Statistical confidence in result (i.e., probability that all links were found)

Algorithm. Define an ***observation*** at time *t*, to be the state of a given set of paths (either congested or not congested). An observation may include all paths or a subset of the paths. Define a vector ***v*** of paths to describe a particular observation, for example

$$v = ab\bar{c}d\bar{g}k.$$

The bar notation indicates that paths a, b, d, k are "on", paths c and g are "off", and paths not included in the list are not constrained (i.e., "don't care"). Then an observation $O_v(t)$ is defined as the event that at time *t*, the monitors observe congestion on all paths marked "on", and observe no congestion on the paths that are marked "off". If all paths are included (as on or off) in the vector, we call it a "full" vector or "full" observation. For any "partial" observation (i.e., one that is not full), there will be many "full" observations that are consistent with it. We may either spell out the vector or not, e.g., $O_v(t)$ or $O_{ab\bar{c}d\bar{g}k}(t)$, depending on the context.

Define a ***trigger*** as a potential cause of congestion, that is, a single link, or a set of links. We formulate a *trigger event* using a similar vector notation (although the interpretation of the vector is slightly different here).

$$T_v(t) = T_{ab\bar{c}d\bar{g}k}(t)$$

is a trigger event at time t that *causes* congestion on paths a, b, d and k, does not cause congestion on path c and g, and may or may not cause congestion on the paths not included in the vector. A trigger vector that corresponds to an actual link or set of links in the network is a "true" trigger, and one for which no such combination of links exists is a "false" trigger. A single link is a true trigger, and a true trigger with a fully specified vector is always a single link (in a reduced graph). Now, solving the topology is equivalent to finding all of the fully specified true triggers, since the vectors identify which paths cross each link.

We construct a probability model that relates the observations to the triggers. Hypothetical partial triggers are constructed and checked against the observations. A trigger that has a high probability of occurrence based on the observations, will lead to one or more true links, while one with low probability will allow us to prune the space of potential triggers, and reduce the computation time.

Define an ***observation probability*** P_v^O as the probability that a specified set of path delays are above or below a given threshold (i.e., congested or not congested). For example,

$$P_v^O = P^O_{ab\bar{c}d\bar{g}k} = \text{Pr[paths a, b, d are congested and paths c, g are not congested].}$$

Observation probabilities can be estimated by the relative frequencies of congestion observed on the various combinations of paths in our discrete time model. Assuming stationarity, the longer the time series from simulation (or measurement), the better the estimate of $\{P_v^O\}$.

Define a ***trigger probability*** P_v^t as the probability that a (hypothetical) trigger is congested, where the trigger vector specifies a set of paths that the trigger either does or does not congest.

$$P_v^t = P^t_{ab\bar{c}d\bar{g}k} = \text{Pr[at least one link crossed by paths a, b, d and not crossed by paths c, g, is congested]}$$

There are many ways to relate the observation probabilities to the trigger probabilities. Here is one system involving two paths:

$$\begin{aligned} P^o_{ab} &= P^t_{ab} + (1-P^t_{ab})P^t_{\bar{a}b}P^t_{a\bar{b}} \\ P^o_{a\bar{b}} &= (1-P^t_{ab})P^t_{a\bar{b}}(1-P^t_{\bar{a}b}) \\ P^o_{\bar{a}b} &= (1-P^t_{ab})P^t_{\bar{a}b}(1-P^t_{a\bar{b}}) \end{aligned}$$

These equations capture the fact that an observation can always be caused by a trigger with the same vector. In the first equation, there is also a combination of the other two triggers that can cause the observation. In the second and third equations, the other triggers (with vectors different from the observation) must be off for the observation to occur. The triggers are a way of partitioning the set of links in the graph. For example, considering two paths, T_{ab} is the set of links on both path *a* and path *b*, $T_{\bar{a}b}$ is the set of links on path *b* and not on path *a*, and $T_{a\bar{b}}$ is the set of links on path *a* but not on path *b*, and $T_{\bar{a}\bar{b}}$ is the set of links on neither path *a* nor path *b*. If the vector is fully specified, then the "all-off" vector cannot be a true trigger (in a reduced graph) because it corresponds to a link crossed by no paths. The all-off partial triggers, however, do play an important role in our general solution, described below. This derivation and notation follows from and generalizes an analysis by Ratnasamy and McCanne [Ratn00].

Now, organize the triggers in a tree that progressively adds one path at each level, as shown in Figure 32. Each trigger is a partition of the links in the network graph. On each level, all of the links are contained in the triggers, and no link appears in more than one trigger. On the last level, where the triggers are fully specified, each trigger contains exactly zero or one link.

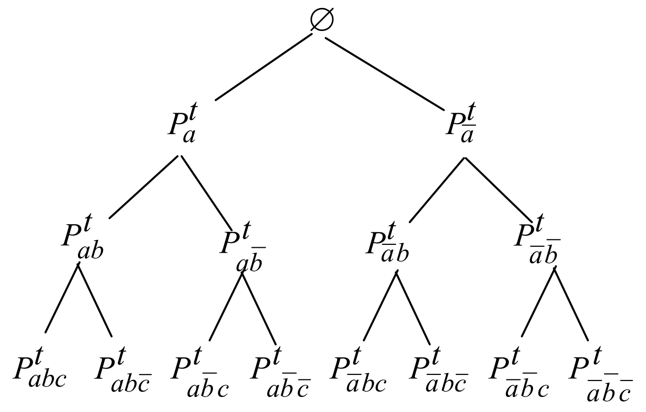


Figure 32 Tree of trigger probabilities. Each level adds one path to the vector specifying the trigger. The bottom level has fully specified triggers. The trigger probabilities that survive pruning to the bottom level will correspond to links in the graph.

We can now describe a system of relationships between triggers and observations that has some very nice properties. We will illustrate the equations for level 3, but describe their behavior for the general case. Note that the equations for a given level are always the same, regardless of how many levels the tree actually has. First, we have the equation that says the observation of no congestion on any path requires that no triggers be on:

$$P^o_{\bar a\bar b\bar c} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{a\bar b c})(1-P^t_{\bar a bc})(1-P^t_{a\bar b\bar c})(1-P^t_{\bar a b\bar c})(1-P^t_{\bar a\bar b c}) \quad (1)$$

Next consider the equations for two observed paths being off (and the remaining one don't care). In the first equation, all the triggers are included that would cause path *a* or path *b* to be on. The others follow in parallel.

$$P^o_{\bar a\bar b} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{a\bar b c})(1-P^t_{\bar a bc})(1-P^t_{a\bar b\bar c})(1-P^t_{\bar a b\bar c}) \quad (2)$$
$$P^o_{\bar a\bar c} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{a\bar b c})(1-P^t_{\bar a bc})(1-P^t_{a\bar b\bar c})(1-P^t_{\bar a\bar b c}) \quad (3)$$
$$P^o_{\bar b\bar c} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{a\bar b c})(1-P^t_{\bar a bc})(1-P^t_{\bar a b\bar c})(1-P^t_{\bar a\bar b c}) \quad (4)$$

Now we can solve for three of the level-3 trigger probabilities by dividing equation (1) by each of the equations in this group. Similarly, we consider the next group of equations, with one less path in the observation probability vectors:

$$P^o_{\bar a} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{a\bar b c})(1-P^t_{a\bar b\bar c}) \quad (5)$$
$$P^o_{\bar b} = (1-P^t_{abc})(1-P^t_{ab\bar c})(1-P^t_{\bar a bc})(1-P^t_{\bar a b\bar c}) \quad (6)$$
$$P^o_{\bar c} = (1-P^t_{abc})(1-P^t_{a\bar b c})(1-P^t_{\bar a bc})(1-P^t_{\bar a\bar b c}) \quad (7)$$

Dividing equation (1) by each of these equations, and substituting the known trigger probabilities, we find three more. The last one P^t_{abc} can now be found from equation (1) and the six known trigger probabilities.

This system has a semi-triangular form, meaning that the trigger probabilities on a particular level in the tree can be grouped and ordered so that each depends only on one new equation, and values of probabilities computed in previous groups. Each groups includes trigger probabilities with the same number of *on* and *off* paths, and the groups are evaluated in order of increasing number of *on* paths. Probabilities within a group do not depend on each other.

At subsequent levels in the tree, the equations follow the same general pattern, although the number of variables and equations increases. If we had to find all potential trigger probabilities, then the size of the system would increase exponentially (doubling with each new path). However, the triggers do not all correspond to actual links in the graph, so we devise a method of pruning out the false triggers at each level. Pruning a trigger reduces the computation substantially, since it deletes all of its descendents on subsequent levels. A pruned trigger probability is set to zero wherever it occurs in an equation.

The system has a nice property that allows us to avoid keeping track of what elements have been pruned out. Each trigger has one parent trigger on the previous level, and if not pruned, two children on the next level. So we can track the surviving triggers as they descend from one level to the next, and the group they belong to in the system of equations is given by the number of paths turned on in the vector.

To prune a trigger, we need to do a discrimination between the ones corresponding to actual links and those that do not. A *true* trigger probability will have a value that is an estimate of the congestion probability on the link or set of links that the trigger corresponds to. A false trigger probability will have a value of about zero. There is some noise in the estimate of the false trigger probabilities because the observation probabilities are an empirically-based estimate of the actual system probabilities.

Discrimination algorithm for pruning. At each level, we compute the trigger probabilities, excluding any that were pruned at a higher level in the tree. We then collect the values into two populations, by ordering them and assigning them to the high (true) population and low (false) population, working from both ends of the ordered list. After some number (e.g. half) of the triggers are grouped, we assign subsequent triggers to groups based on which population the trigger probability is closer to, in terms of the mean and standard deviation of the groups. After grouping, we chose a threshold between the two groups at the point τ that is an equal number η of standard deviations from the mean of each population. That is,

$$\eta = \frac{(\tau - \mu_1)}{\sigma_1} = \frac{(\mu_2 - \tau)}{\sigma_2}$$

$$\tau = \frac{(\mu_1\sigma_2 + \mu_2\sigma_1)}{\sigma_1 + \sigma_2}.$$

If we assume that the distributions are close to Gaussian, then η can be interpreted as the number of standard deviations of the distribution of true triggers we are capturing by the threshold. So, $\eta = 1$ means an 18% chance of missing a link, $\eta = 2$ gives a 2.5% chance of missing one, $\eta = 3$ a 0.5% chance, etc. (Whether the distributions are well enough behaved that these probabilities are accurate remains to be shown, but in any case, η provides at least a heuristic goodness measure for the topology discovered in this algorithm.)

If we discriminate the true and false triggers at each level, then the number of active triggers will only increase until we have as many as the true links in the network graph. These will descend, through the tree, doubling at each new level, and then pruning out half before going on to the next level. At the last level, the vectors will be fully specified, we will have as many true triggers as links in the graph, and we will know the identity of the paths traversing each link in the graph.

At this writing, we are still evaluating how large a network we can discover with this method. With about 30 Mbytes in RAM memory, and allowing 4 hours of CPU time, we can discover the network with 19 monitors shown in Figure 33. In this case, the simulation is run with the condition that links are congested about 10% of the time. If the link congestion rate is arbitrarily low, we will not be able to discriminate the links as true triggers. There is a tradeoff at the edges of the feasible parameter space between the length of the time series, the confidence in the accuracy of the result (the parameter η), the simulation time, and the size of the network discovered.

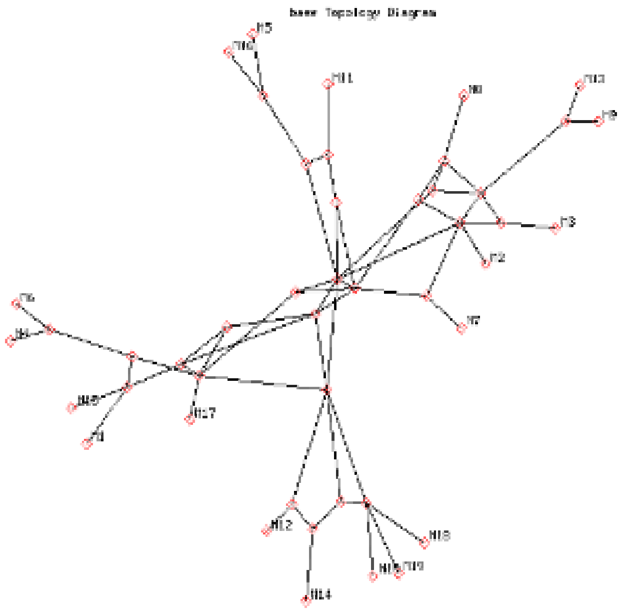


Figure 33 Example topology discovered using the Cross-correlation method case 2 algorithm. This network has 19 monitors, 93 links and 342 paths. (The Matroid method was used to derive the graph from the path-component list that was output from the Cross-correlation method.

6.5 Matroid Method - from path component list to graph

6.5.1 Problem Statement

There is an unknown network (its links are all directed), that we would like to reconstruct as far as possible. The information we are given is as follows. We are told that there are certain special nodes of the network, called "monitors", and we are told how many monitors there are and what their names are. In addition we are given some paths between monitors. More precisely, we are given a number of distinct ordered pairs of distinct monitors, and for each such pair, say (u,v), we are given the set of links that lie in some directed path of the network from u to v. We are NOT given the order of the links in the path. In addition, it is guaranteed by the user that

(i) for each such set of links, the path it makes is "simple", that is, it has no repeated nodes or links

(ii) for any two of the paths that happen to be going to the same monitor, the set of links they have in common is also a path going to the same monitor

(iii) no monitor is an internal node of any of the paths.

If these conditions are not satisfied, the program will return garbage, or possibly notice the error and complain. (I think it won't crash, even with garbage input, but it might.)

6.5.2 Input Format

The program is called by a command line call "mat <problemfile>", where <problemfile> is the name of the file containing the set of paths. The problemfile should look like this:

```
80 paths

20 monitors

P1 M11 M4 3.142 5 L212 L353 L115 L62 L302

P2 M7 M13 3.142 4 L322 L300 L26 L64

P3 M10 M18 3.142 4 L58 L312 L205 L277

P4 M6 M2 3.142 3 L190 L94 L318

......
```

i.e., the first line gives the total number of paths described in the file, the second is the total number of monitors, and then there is a line for each path. Take the first path above for instance. The line starts with "P1", which is the name of this path.

WARNING: the first path in the file MUST be called P1, the second P2 and so on.

The second and third entries M11 and M4, are the names of the monitors that this path starts from and goes to, respectively.

*WARNING: The names of the monitors MUST be M1, M2, and so on (no gaps!). *

The fourth entry 3.142 is a junk real number, not used by this program but included for compatibility with some other programs that need to read this file. The fifth (5 in this case) is the number of links in this path.

Then there follow the names of these links.

WARNING: The names of the links MUST all be of the form "Lx", where x is a positive integer.

6.5.3 Output Format

There are two sets of output files, called output0, .., output3, and cloud1,.., cloud3. As the algorithm progresses, we reconstruct the network more and more, but in the later stages with some (progressively wilder) guesswork (which could be viewed as making stronger and stronger assumptions about the network). The file output0 contains just the input data, sorted and (optionally, via a #define) with any series links suppressed. After stage 1 of the algorithm (which makes no guesses), our current knowledge is summarized in the files cloud1 and output1. The file cloud1 contains what we have been able to reconstruct of the network in the standard form of a FELIX network topology file. output1 contains our updated knowledge of the sets of paths (we order the links of each path as much as we can; thus, we might know that in path P23, some given three links come first (in some order), followed by some other given four links (in some order). Similarly at stages 2 and 3 we output our current knowledge.

The three "cloud" files are intended to describe the network partially; but to do so, they describe another network completely. This second network is related to the first by

(a) omitting any links that do not occur in any of the given paths (clearly we have no chance of learning about those links)

(b) replacing the portions of the network that we have been so far unable to reconstruct by "clouds". (Each cloud is represented in the file by a single fake node, adjacent to the nodes of the network that lie on the boundary of the cloud.)

When we have (partially) reconstructed a network, we may have deduced that there is a node incident with some given set of links, but, while we know the real names of the links, we have no way to find the real name of the node - it isn't in the input to the program. So we have to invent a name for this node. All new nodes (not monitors) are given names like N77, N followed by an integer larger than the number of monitors. (This list of names may have gaps - this is because at times the program realizes that two nodes it thought were distinct are actually the same, and it identifies them and removes one of the two names, possibly leaving a gap. We didn't bother to close up the ranks.) Links names in this file may consist of one of the original link names, or may consist of several in the form (say) L131/L254/L127/L204; this latter means that the program detected that these four original links were in precisely the same input paths, and they were combined to make one composite link.

The fake nodes that represent clouds are given names C1, C2 and so on; and the fake links from a "cloud node" to a real node are called names like LC3M17 (meaning this link is from cloud C3 to node M17, which is a monitor because it begins with M).

At the end of the cloud file is a list, for each cloud, of the names of the links that are hiding in that cloud somewhere. These lines are commented out because the current FELIX network topology file format does not recognize them.

6.5.4 Key Ideas Of The Algorithm

There are several steps in the algorithm. The first is:

1 - Tree reconstruction.

The user promises that for each monitor, the input paths going to that monitor have the property that whenever two of them meet, they stay together until they reach the monitor; that is, the union of these paths is a tree. The first step of the algorithm is to reconstruct this tree, for each monitor. More precisely, for each monitor M we find a tree T, so that

(1) M is a node of T, and so is every monitor M' so that one of the input paths is from M' to M, and all the latter are leaves of T (vertices of degree 1) (2) every leaf of T (except possibly M itself) is a monitor M' so that some input path is from M' to M

(3) every link of T represents one or more links of the network, and no two links of T represent the same link of the network

(4) for every input path P from say M' to M, the path P' of T from M' to M has the property that the links of P are precisely those represented by the links of P'.

If such a tree exists it is unique and easy to find. It does NOT exist if and only if there are three input paths (say P,Q and R) all entering M, so that some link belongs to P, Q and not R, and some other link belongs to Q, R and not P. If this condition is violated by the input the algorithm will detect it, and output the three offending paths with a complaint (in the function "failparse"). Otherwise, we find the tree (in the function "parse"). This part of the program uses a separate data structure "treenode" (it doesn't really need to, and probably should be rewritten to use the same data structure as the rest of the program).

As the algorithm proceeds, we shall represent our current partial knowledge of the network in two forms. The first form is easy to describe, as follows. We shall learn that some of the input paths are composed of several pieces (shorter paths) IN ORDER, and we shall know the order, and we shall know the links of each piece. Also we shall know the names of the ends of these pieces (nodes that are not monitors have names we make up). As we go further these pieces will themselves be broken up into chains of smaller pieces.

So for instance, having found the tree T for monitor M as described above, we can break up each of the input paths that arrive at M into several pieces, one for each of the links of T that it traverses. Indeed, the whole tree T is evidently composed of these same pieces, and it turns out more convenient to remember the decomposition of T (rather than remember the decomposition of each path, which would involve a good deal of redundancy). However, sometimes we need to quickly read off how a given input path is decomposed into pieces, so we designed the description of the tree-decomposition to make this possible. Each node (including all possible future nodes that we have not yet discovered) has a number, not too big (we will never get past the total number of links plus monitors, for instance), so we describe the tree, by a map "step"; step[v] is NULL if the node with number v does not belong to the tree, and otherwise it is the piece of the tree starting from node v heading towards the root. This makes it easy to read off the constituent pieces and nodes for each of the input paths (for the input path from M6 to M, first look at step[6], find the head (say N9) of that piece, look at step[9] and so on).

So that is our first way of describing our partial knowledge of the network. (We call this the "tree data-structure".)

At this initial stage we have a tree for each monitor, but the trees are completely node-disjoint except for monitors. As the algorithm proceeds, we shall sometimes discover that a node of one tree and a node of another are really the same node of the network, and then we shall identify them in the tree data-structure. So in general, a given node may belong to several of the trees.

2 - Representations.

For reasons that we will explain later, we would like to assign a binary vector to each link, satisfying certain conditions. Here we explain the conditions and how we find the vectors. First, “binary vector” means a vector of 0’s and 1’s with arithmetic modulo 2, so 1 + 1 = 0.

Say a set X of links of a network is "Eulerian" if for every node of the network, the number of links in X incident with this node is even (possibly zero). So for instance the edge set of any cycle is Eulerian. (Link directions are ignored in this definition.) Let us (for convenience in this description; we don't really do it in the program) add to the network a new node S say, and undirected links between S and every monitor. A "representation" means a choice of some binary vector for every link (including the links to S), so that for every non-Eulerian set X of links the vectors of the links in X have non-zero sum. The converse statement (that for Eulerian X the sum is zero) does not have to be true for all X, but it needs to be true quite often, in order for the representation to be of any use.

Now each of the input paths can be augmented by adding two of the new links to it to make a cycle through S; and we know the link sets of these cycles, because we know the ends of each of the input paths. Call these cycles "input cycles". We can use the input cycles to find a useful representation, as we explain below.

Given a binary matrix A, it is easy linear algebra to find a binary matrix B so that for any binary vector x of the right length, Bx = 0 if and only if there is a binary vector so that Ay = x. (Basically you do row operations on A to make a set of columns be the identity matrix, discarding dependent rows, and then transpose the non-identity part of the result, and combine it with another identity.)

So, take the rows of A to be the input cycles (its columns are indexed by the links). The vectors we are seeking are the columns of the matrix B. These have the property that for a set X of links, the vectors sum to zero in X if and only if X can be constructed from the set of input cycles by symmetric difference (i.e., X is in the linear span of the input cycles); and since the input cycles are Eulerian, and symmetric difference preserves this property, it follows that any such X is Eulerian. (This is carried out in the function "findrep" of the program.)

Given a representation, let us say a set of links is "balanced" if the sum of the vectors of the links in the set is zero. When we apply this later, we shall use it to detect that certain sets of edges are Eulerian, because they are balanced. The method will fail to recognize (as Eulerian) sets which really are Eulerian but which cannot be expressed as a symmetric difference of the input cycles. (Not quite true - we shall occasionally be able to improve the representation, so that it can recognize more Eulerian sets, but this is a rare occurrence.) Thus we would prefer there to be many input cycles, to reduce the likelihood of failing to recognize Eulerian sets. In practice this is reasonably effective; it seems that the linear span of the input cycles is usually pretty close to the linear span of ALL the cycles of the network. However, there is often a gap between the two, and this is where the "clouds" are born.

While it is not true in general that every cycle is balanced, we will arrange the representation so that all cycles that we know about are balanced. From time to time we will find a new cycle (i.e., not in the vector space spanned by the cycles we already discovered) and this will enable us to find a better choice of the representation; and consequently we are slowly able to reduce the dimension of the vector space from which our vectors are chosen. (This is done in the function "insertcycle".)

Incidentally, that dimension is always at least the total number of nodes (monitors and discovered nodes, but not counting S), and we get equality only for the "ideal" choice of vectors (which will recognize all Eulerian sets for us), so the difference of the two gives a concrete measure of how good the method is in any particular case.

So, that is our second way to describe partial knowledge of the network - we retain a "representation", a binary vector for each link, so that they have non-zero sum on any non-Eulerian set of links. We call this the "vector data-structure". We might as well augment this a little. For every node v of the tree data-structure, choose a path from v to S, and sum the vectors of the links in this path; the result we call the "vector of v". It is independent of the choice of path, since all known cycles are balanced.

3 - Identifying nodes

We have so far found the tree formed by the paths entering each monitor, and in general these trees have many internal nodes, but so far we have no way of telling whether a certain node of one tree is the same node as some node in another tree. Here is what we do. First we find a representation, as explained above.

Suppose that the vector of some node v is in the linear span of the vectors of the links in some tree T (of the tree data-structure). This tells us nothing if v already is a node of T, but if not then it tells us that v SHOULD belong to T. It does not necessarily have to be identified with one of the nodes of T, however; it may be that v is supposed to live in the middle of one of the links of T - more precisely, that for some link of T, the piece it represents is a path of the network which passes through the node that v represents. In this case the piece can be broken into two shorter pieces, which v an end of both of them. It is routine to identify which piece should be broken in two, and what the correct partition is.

To check whether v should be added to T, we do row operations on the representation to make the vectors of the links in T be identity vectors (in the function "standardform"), and since that is time-consuming, we check at this stage all possible nodes v. (This is done in the functions "addvertstotree" and "fixnewverts".) To avoid rechecking the same nodes later (when this function is called again) we classify nodes into two kinds, "VERTEX" and "NEWVERTEX", and only check those called the latter (and change their classification to "VERTEX" when we are done (for all T). If ever we improve the representation, all nodes have to be checked over again, so they are all reclassified as "NEWVERTEX". Occasionally we discover a new node by subdividing a piece, and that would be classified as NEWVERTEX until it has passed through the "fixnewverts" machinery just described.

4 - Identifying pieces.

It might happen that there are two pieces in the tree data-structure which have precisely the same set of links. (After all, pieces from different trees often share links, though pieces in the same tree do not, of course.) In this case, the heads of the two pieces are equal, and so are their tails, and we might as well identify them, and identify the two pieces. To do this we have to update both the tree data-structure and the representation. (This is all done in "differentpiece".)

The modification to the first is obvious. For the second, since we have learned that a pair (indeed, two pairs) v,v' of nodes are equal, we can deduce that a new set X of links is eulerian (even though the vectors in X do not sum to zero); and we can use this to improve our vectors (reducing the dimension of the vector space). (Done in "makeparallel".) Having updated the data-structures, now we run the earlier routines again, in the hope of further identification.

This ends stage 1 of the algorithm. Up to here, everything we have reconstructed about the network is correct; we have not made any guesses or assumptions, except that the paths entering each monitor form a tree, and this is supposed to be true. The current state of play is output in cloud1 and output1.

Now we begin stage 2.

5 - Finding Y-structures.

We search among all pairs of pieces in the tree data-structure, and see if there are two pieces with the same head or tail v, and with at least one link in common. If we find such a pair we make the assumption that the set of links they have in common makes a subpath starting (or ending) at v; and this allows us to subdivide both pieces into two smaller pieces, this common subpath and one other. (Note - both pieces MUST have a link not in the other, for if one were a subpath of the other we would have spotted it by applying "fixnewverts".) If this succeeds we have to modify the tree data-structure, but not the representation. We try all possible pairs of pieces (in "wyesearch"), and if we get some successes we return and rerun all the earlier methods. If in turn one of those give an improvement we run "wyesearch" again, and so on. When there is no further improvement we output cloud2 and output2, and begin stage 3.

Note that we made an assumption about the network, that if two pieces share a link and have the same head (or tail), then their union is a tree. This might not be true, and if not we will fail to find the right network. We might find another network (and in this case we have no way to tell this is not the right answer), or we might deduce that no network exists with the properties we are looking for (in which case the program terminates with a complaint about "graph anomalies").

6 - Finding intersections of pieces.

In stage 3 we make an even stronger assumption; that for any two pieces (still of length >1) that share an internal node, their union is a tree. (Actually we don't need quite so much, but it is complicated to state the precise assumption.) The representation will sometimes tell us that an internal vertex of a piece lies in a tree (not containing the piece). Then we have two pieces that we know meet, but we don't know how they meet. Our assumption is that they meet in the simplest way that seems consistent with what we know. (This is done in the function "matroid".) If this is successful we return and run all the earlier methods again, and if that gives an improvement we rerun "matroid", and so on.

7 - Monovalent monitors

There is, optionally (controlled by "#define MONITORLEAF" in the program), sandwiched between stage 2 and stage 3, a place to insert the information that every monitor has only one neighbor (out or in); or it can be varied to say the same except for monitors where this is patently not true, or it can be turned off. If turned on, this may result in identifying pairs of nodes that were not known to be the same previously, and in this case we rerun the methods of stage 1 and 2 again. The output at the end of stage 2 includes any improvements made by this function (called "leafmon").

8 - Finding the clouds

At the end of stage 1, 2 and 3, there may still be some pieces in the tree data-structure with length > 1, in which case we have not reconstructed the network completely; we still don't know the order of the links in any such piece. One reason might be that there are two different networks both consistent with the input data, and we can't tell which is the right one. (Or the algorithm is not clever enough to work it out; or we have made a previous assumption which was false; or the input was garbage.) In any case, it turns out that when there is one problem piece there is a group of them, with overlapping link sets; and none of the links in these problem pieces lies in any piece we understand (i.e., of length 1). This overlapping group of pieces is replaced by one big set (the union of the link sets of the pieces) called a "cloud", for the output at the end of each stage. No two clouds share links, though they might share nodes (because there might be two problem pieces at some node which belong to different clouds).

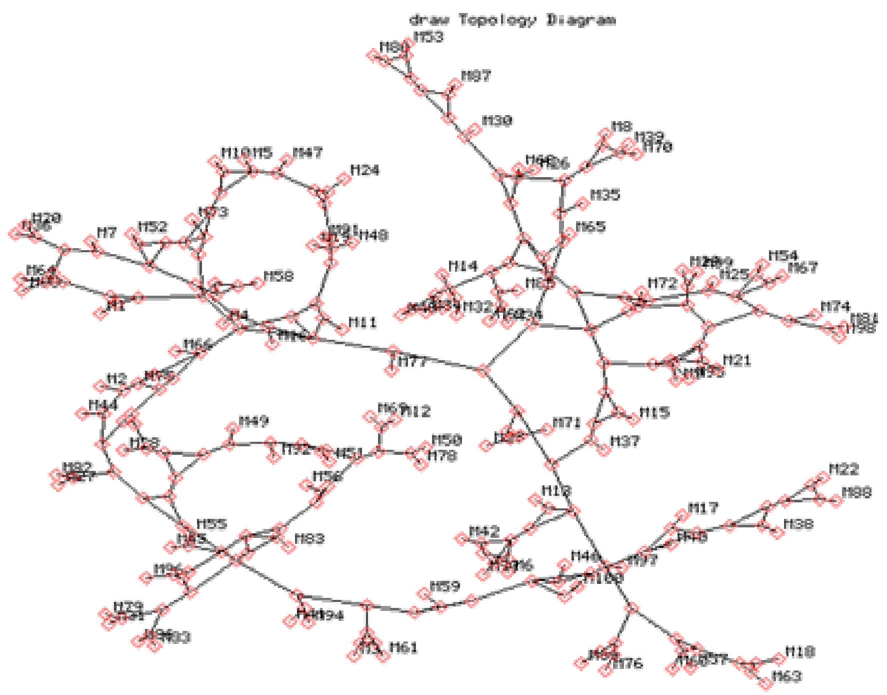


Figure 34 Example topology derived from (unordered) path-link data by our Matroid method implementation. Network contains 100 monitors, 187 internal nodes and 698 (unidirectional) links. The discovered network contains 2 clouds (appearing here as internal nodes), each with 3 links that could not be ordered.

6.6 Distance Realization Method

Another topology realization method (attempting solution F from Figure 3) is based on distance matrix realization. This problem is, given a symmetric, nonnegative matrix **M** that satisfies the triangle inequality ($\mathbf{m_{ij}} + \mathbf{m_{jk}} \geq \mathbf{m_{ik}}$), find an undirected graph **G**, with nonnegative lengths assigned to its edges, so that **M** is the distance matrix between some of **G**'s nodes. More particularly, we look at the problem of finding a distance matrix realization of minimum total edge length.

This relates to the Felix topology realization problem if we consider **M** to be the matrix of monitor-to-monitor minimum total delays, and we want to find a graph with delays assigned to each link, so that the sum of the delays along the monitor-to-monitor route will equal the entry in **M**. We must make several simplifying assumptions.

1. We assume that the delays on links are symmetric. This is probably a bad assumption if we include queuing delay, but not so bad if we take the minimum delays, assumed to be transmission delays only. This also requires that all links be bi-directional. This assumption can probably be dropped, increasing both the complexity of the algorithms, and the multiplicity of solutions (see the tree-growing method on this).
2. We assume that the routing is minimum-delay routing. The tree-growing method managed to avoid this issue, but in a general graph we have to worry about routing.
3. We implicitly assume that the matrix of delays is accurate.

We choose the graph of minimum total edge length to try to select the simplest possible solution that fits the data, and avoid overly dense graphs. We want to avoid having a complete graph, with a direct link between each pair of monitors, which can fit any minimum delay matrix. It can be shown that if there is a realization that has a bottleneck edge, a link over which all traffic from one set of two more monitors to the remaining set of two or monitors must pass, then the minimum total length realization will have a bottleneck edge between the same two sets of monitors.

The problem of finding a minimum total length distance matrix realization has been previously studied, and found to be NP-hard. Fan Chung of Telcordia and UCSD has recently discovered that the problem is also very sensitive to its data, in that if a matrix **M** has a sparse realization **G**, there will be arbitrarily close matrices **M'** whose realizations must be substantially denser, and have total length greater by a factor on the order of the number of monitors.

We put together and coded a heuristic algorithm for this problem based on the idea of local improvement. An initial graph is built, with edges between each pair of monitors, each of the specified length. In a first phase, all triangles in the graph are successively replaced by 'Y'-shapes, with edge lengths chosen to maintain the required distances, but half the total length of the triangle. In the second phase, any 'V' of two edges incident on a common node who do not lie in a common shortest path between monitors are replaced by a 'Y' of lesser total length. In a third phase, any edges that can be removed without increasing distances between monitors are removed. If the third phase makes any changes, the second and third phases are repeated until no more changes are made.

We deal with the sensitivity to the data in two ways. First, throughout the algorithm we maintain an "epsilon" value, below which numbers are treated as zero. This mainly is to reduce problems with rounding error in the algorithm. Second, we allow the user to specify a clipping threshold, so that any final edge shorter than this threshold will be contracted to a point.

Experiments using this algorithm on minimum delay matrices between monitors on simulated internets, where we have forced symmetric delays and minimum-delay routing, show the following results.

1. As expected, bottleneck edges (cut-edges) are found.
2. Triangles are never found, as they can easily be modified to produce a shorted graph.
3. Isolated 4-cycles are found.
4. Sets of two edges that together join one group of monitors to the rest are sometimes found.
5. More complicated structures are not reliable identified, probably both because the algorithm is only a heuristic, and because the actual network is not a minimum-length realization itself.

In response to the sensitivity problem, Fan Chung came up with the following two variant problems, which are more robust.

1. Weak realizations. A weak realization of a distance matrix is a connected graph with edge lengths, so that the distance between two monitors in the graph is at least the value specified in the matrix. In this context what we have been calling a realization will be called a strong realization. Any connected subgraph of a strong realization that contains all the monitor nodes will be a weak realization. A minimum total length weak realization will necessarily be a tree. If two matrices differ by only a small amount, then the lengths of their minimum length weak realizations will differ by only a small amount also. Finding a minimum length weak realization is still NP-hard. We have a heuristic for this problem based on the local improvement method for strong realization but starting with a minimum spanning tree,

followed by a more general 1-opt idea of testing the results of replacing arbitrary edges and re-optimizing.

2. Rooted weak realizations. A rooted weak realization of a distance matrix is a weak realization of the matrix with a root node, so that the distances of other monitors to that root node are exactly as specified in the matrix. The union of a choice of shortest paths from each monitor to a root monitor will be a rooted weak realization. A minimum total length rooted weak realization will be a tree. Finding minimum length rooted weak realizations is NP-hard. We have a heuristic for this, based on the same ideas as the unrooted weak realization heuristic, but starting with a star from the root.

These problems are of mathematical interest as more robust variants of the strong realization problem, but they can also fit into the topology discovery problem, at least under the earlier assumptions of this section. Suppose we find a near-minimum length rooted weak realization for each monitor as root. The rooted weak realization heuristic is more exhaustive than the strong realization heuristic, because it is more efficient to work on trees. Perhaps we use other information such as from the cross-correlation method in trying to make these weak realizations resemble the trees of paths with a common destination. We can try to merge these realizations into a single strong realization and hope that this will be better than the results of the strong realization heuristic. We are still experimenting as to how best to merge the weak realizations. One simple method is to take the union of the weak realizations, identifying the monitor points from the various weak realizations, but keeping the internal nodes separate, and then running the local optimization of the strong realization heuristic on the resulting graph. This doesn't produce consistently better or worse results than the strong realization heuristic itself.

The strong realization algorithm is embodied in the program "realize". The weak and rooted weak realization algorithms are embodied in the program "weak". The current algorithm to merge rooted weak realizations into a strong realization is embodied in the program *tree_merge*.

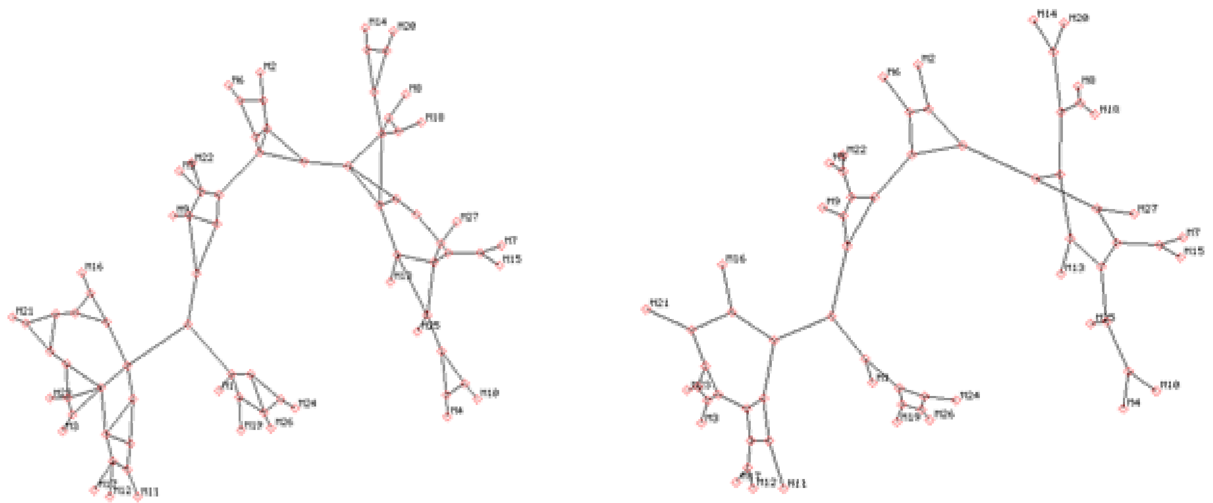


Figure 35 Example original reduced network graph and the corresponding graph discovered by the Distance Realization method.

7. Graphing and mapping

7.1 Graph rendering tools

For the purpose of rendering, a network can be described by a list of nodes (monitors and interior nodes), a list of the edges that connect the nodes, and, optionally, the locations of some or all of the nodes. This information is a description of a graph. The graph is rendered by placing the nodes in two dimensions and then drawing the edges. We have written a C program called *topomap*, based on the algorithms used in Javasoft's example GraphLayout applet, that produces a rendering of a given network description. The program uses an iterative algorithm that places the nodes by attempting to push nodes apart while also trying to keep the edges at a particular length. The algorithm works best for graphs that are trees, but also does a remarkable job at rendering more general graphs (see Figure 36). One way to usefully render the network is to place the monitors and important interior nodes (e.g., NAPs) in their correct geographic positions and overlay the layout on a map (e.g., Figure 37). However, when nodes are crowded into a small area, it may be better to fix some nodes geographically and let others float in the general area.

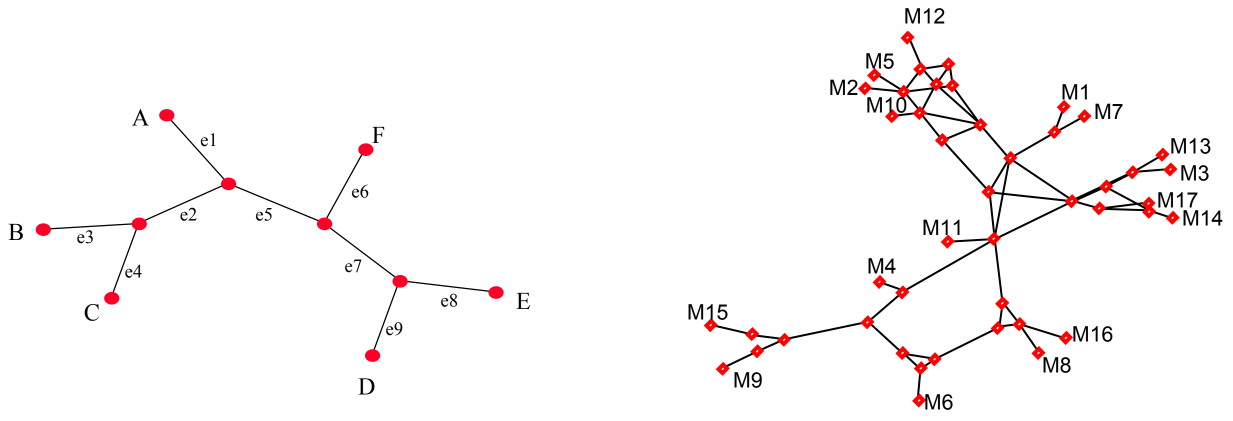


Figure 36. Connectivity maps rendered with topomap: a tree network, and a more general mesh network.

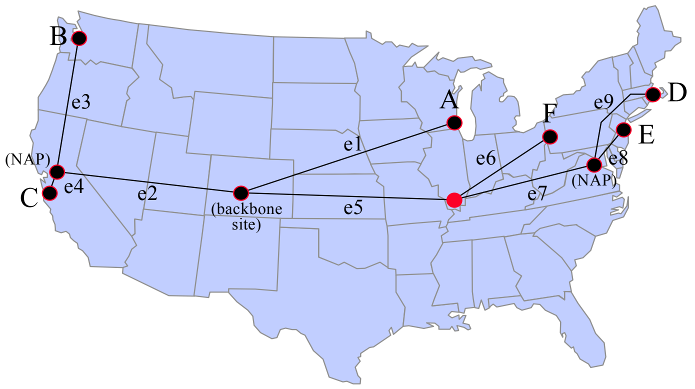


Figure 37. Geographically accurate map. Monitors and well-known nodes are placed at the correct locations.

7.2 Traceroute interpretation tools

Traceroute is a tool that gives the forward path between the source host (the host on which traceroute is executed) and a specified destination host. Example traceroute output is shown in Figure 38. The hosts/routers along the path are given by name (if available) and IP address, the second and third fields of the numbered lines of output. The source host is not listed.

```
traceroute to brooklyn (192.4.18.61) 30 hops max, 40 byte packets
1  128.96.73.254 (128.96.73.254)  4 ms  2 ms  3 ms
2  128.96.215.253 (128.96.215.253)  3 ms  3 ms  4 ms
3  pya-wanhub-cisco.cc.bellcore.com (128.96.250.12)  4 ms  5 ms  6 ms
4  128.96.44.254 (128.96.44.254)  6 ms  8 ms  31 ms
5  lab214b-cisco.cc.bellcore.com (128.96.34.40)  14 ms  8 ms  6 ms
6  celebrator (192.4.14.15)  8 ms  8 ms  26 ms
7  brooklyn (192.4.18.61)  14 ms  8 ms  8 ms
```

Figure 38. traceroute from buzzard to brooklyn.

We have created several tools for manipulating the information provided by traceroute. The *tracepath* program traces paths from the source host to a list of hosts and summarizes them in a single file (Figure 39).

```
Filetype:      Felix Path File
Version:       1.0
Generated:     by tracepath (bss) on Thu Jul  8 14:23:15 1999
Paths:  7
buzzard buzzard : 128.96.73.102 128.96.73.102
buzzard santorini : 128.96.73.102 128.96.73.75
buzzard offkey : 128.96.73.102 128.96.73.254 128.96.215.253 128.96.250.12
128.96.44.254 128.96.34.40 192.4.14.15 192.4.16.66
buzzard bluemoon : 128.96.73.102 128.96.73.254 128.96.215.253 128.96.250.12
128.96.44.254 128.96.34.40 192.4.14.15 128.96.63.10 207.3.231.32
buzzard brooklyn : 128.96.73.102 128.96.73.254 128.96.215.253 128.96.250.12
128.96.44.254 128.96.34.40 192.4.14.15 192.4.18.61
buzzard breeze : 128.96.73.102 128.96.73.254 128.96.215.253 128.96.250.12
128.96.44.254 128.96.34.40 192.4.14.15 192.4.5.9 * 192.4.6.9
buzzard wind : 128.96.73.102 128.96.73.254 128.96.215.253 128.96.250.12
128.96.44.254 128.96.34.40 192.4.14.1 192.4.13.8
```

Figure 39. Path file summarizing paths originating from buzzard.

The *pathconvert* program can convert the path file into various formats:

- A symbol file for the dataset, which lists the host names, host addresses, and a short symbol (A-Z or a-z for monitors; n### for intermediate routers along the paths). The file can be modified by the user to give locations and/or more mnemonic names for the nodes and can then be used as an input to pathconvert or other programs.
- An html file containing data for the Graph applet which does interactive rendering
- A Felix Topology File (which specifies node locations if they were provided in a symbol file used as input). This can be used as input for the *topomap* program which does batch rendering (using the same algorithm as the Graph applet).

- The common component matrix
- The path component matrix
- Reduced/renamed versions of the path file.

Pathconvert can also perform several operations that reduce the traceroute information into more accurate or more easily viewed versions of the network:

- If the –x option is specified, *pathconvert* merge nodes that correspond to the same host name when looked up using DNS. The resulting merged node has the same canonical name and IP address (the first listed) for the host/router.
- If the –y option is specified, *pathconvert* merges nodes that enter a router at the same interface (IP address). For this operation, we assume that the connections between the nodes are point to point (e.g., unidirectional) and not shared (i.e., not an Ethernet subnet). Thus, all connections entering a router at the same interface must have come from the same router (at the other end of the connection). For the source router to have two different IP addresses, the paths must have entered the source router at different interfaces. This operation creates grouped nodes.
- If the –z option is specified, *pathconvert* merges series edges. This operation merges adjacent links and nodes that cannot be distinguished from each other, i.e., where there is no branching at the points where link pairs meet. The algorithm eliminates intermediate nodes that have only two edges.

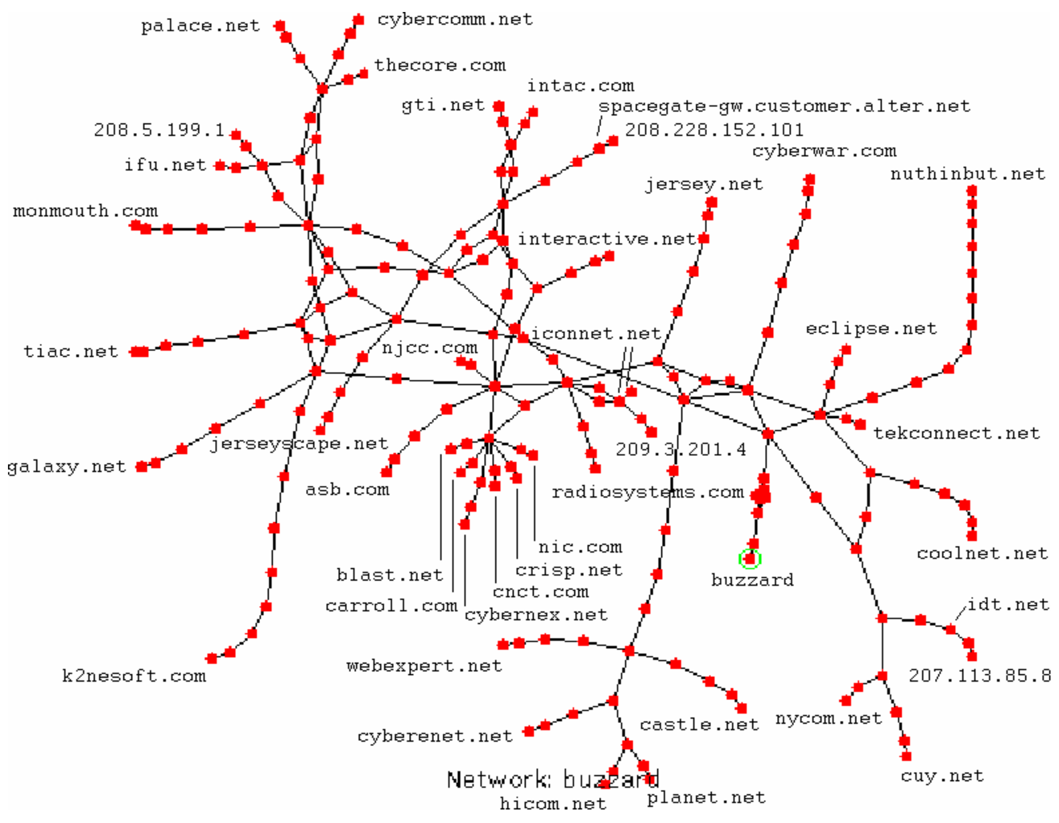


Figure 40 Map of internet in New Jersey created with Felix traceroute tools described in this section.

8. Future work topics

Further development of topology discovery algorithms. The “cross-correlation” method seems particularly promising, and we would like to develop this method further. It is especially valuable, since it complements the “Matroid” method to yield a complete solution, from delay data to network topology graph. The “distance-realization” method is also a good candidate for work, since it contains several promising variations that we have not yet adequately explored.

Performance assessment algorithms. The “spike-tail” method is also promising, and not yet fully explored. We may come up with other methods for assessing link performance to round out and complement the system capabilities. These algorithms can be designed assuming the correct topology is known, to separate the two problems.

Statistical analysis. An important aspect of the cross-correlation method is some statistical compensation of the data to make the analysis more robust against noise. This task will reveal some aspects of one-way delay data in real networks that will be useful independent of the particular method explored.

Representation of graphs and maps. The output of our system is a graphical rendering of the network topology and “health”. A number of interesting issues arise here related to showing partial information in a meaningful way. For example, the location of some monitors, but none of the internal nodes may be known, yet the map can be accurate enough to present a reasonably useful view.

Partially discoverable topologies. A related topic is how to interpret data that leads to only a partial reconstruction of the network. How can the system decide among several equivalently good solutions, or present the ambiguity in a useful manner.

Clock drift correction. A useful extension of out “fixdrift” method will be to take a reasonably large measurement sample from monitors running Network Time Protocol (NTP), but with otherwise significant clock drift, correct the data for drift, and then apply the topology discovery algorithms and see if fixdrift is sufficient to make such data useable. Further research on clock drift compensation would probably be quite productive.

Analysis from packet loss data. This is an obviously useful extension of our analysis from delay data, and may complement that analysis to make a more robust system, overall.

Analysis from throughput (packet pair) data. Another extension of delay-data analysis.

Hybridize analysis with more intrusive methods, e.g., traceroute. Here, the idea is to supplement out measurements with a very small level of more interactive measurements. For example, traceroute could be used to confirm the identity of some key internal nodes.

Security features in monitors. If the Felix monitors themselves are of interest to Darpa, we can spend some effort to make these more robust.

Anomaly detection with Felix-style monitors. This topic could serve to bring our efforts closer to the focus of other projects within the Darpa Intrusion Detection program.

9. System requirements and computing environment

The Felix monitors are written in the C programming language and are currently compiled for both Sun Microsystems Solaris and Linux operating systems. A Makefile is included to aid in compilation. Both the monitors and the archive server are compiled using gcc, which is available for Solaris as well as Linux. The hardware platform for the monitor is typically a Linux personal computer or a Sun Microsystems workstation.

The back end of the system is composed of the archive server, Postgres database management system, and the web server. Our current back end machines are Sun Ultra 5 workstations. The archive server is a C language program and is currently run under the Solaris 2.7 operating system. Postgres is a freeware product and is available from http://www.postgres.org/index.html. The web server could be any of several available as long as it supports CGI script. Our current web server is Apache 1.3.6 and is available from http://www.apache.org/. User queries typically result in some type of image, for example a statistical plot or topology diagram. The means of rendering the images is through the use of GNU Plot. GNU Plot is a generic plotting tool and is available via http://www.cs.dartmouth.edu/gnuplot_info.html

The user interface is implemented in HTML and gives a measure of platform independence and ubiquitous access. We are using Netscape for our browser but it could just as well be any other current browser. The HTML pages contain a series of HTML forms for user option specification. The forms interact with CGI scripts run on the Web server.

Appendix A. Detailed documentation of code

List of all program names:

archsvr	imm	process_mat
corr	mat	real
fan.cgi	matroid	realize
felix.cgi	monsolaris	reduce
fixdrift	pathconvert	regress
fixed	pathfilter	tiers*
fxplot	paul.cgi	topomap
fxsim	pgconvert	tracepath
fxtree	plotall.sh	tree_merge
(gnuplot)*	(postgres)*	weak

*Note: gnuplot is a freeware graph plotting package, available at www.gnuplot.org. postgress is a freeware database package, available at www.postgreSQL.org. tiers is a program from a freeware package to create realistic internet-like network topologies from K. Calvert and E. Zegura at Georgia Tech, available at www.cc.gatech.edu/projects/gtitm [CAL97]. We do not include documentation of these packages (except tiers) in the man pages in this section.

The Felix code is distributed to Darpa as a unix tape archive (.tar) file located at ftp://ftp.telcordia.com/pub/jjd/felix.tar. This documentation file may be found at ftp://ftp.telcordia.com/pub/jjd/felix-project-final.doc. The location for programs given in the SYNOPSIS section of the man pages assumes that the tar file is unpacked in a directory called "felix". For cgi scripts and files accessed through the web server, code may need to be moved to a directory visible to the local web server (our convention is of the form: /u/xxx/public_html/file.)

A.1. archsvr

NAME

archsvr – The archsvr acts as a distributed SQL front end to the DBMS for both the monitors and the graphical user interface.

SYNOPSIS

felix/dbPostgres/archsvr [-p <port number>]

-p <port number> - local port to bind to the server

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

The archive server (AS) is a concurrent server that processes database transactions placed by a web based GUI or the data collection monitors.

The archsvr populates the PostgresSQL DBMS with entries generated by the monitoring stations. Each monitoring station periodically writes its local network view to the archive server which in turn writes the datasets to the DBMS; currently each packet exchange between monitors generates an archive event.

Data analysis requests are placed by users via a web based GUI which is implemented with HTML forms that in turn call CGI scripts. Queries can be for GIF Cartesian plots based on database contents, Cartesian plots of pre-existing files accessible to the AS, topology graphs, or par-wise delay and packet loss statistics in chart form. Plot types include one way, pair-wise time series of delay and packet loss, probability density, autocorrelation, variance time, and log-log distribution. The AS retrieves the necessary data from the DBMS, performs the appropriate statistical manipulation on the data and creates a GIF formatted graphical representation of the data. These gifs are then made available to the calling CGI script which composes an HTML page on the fly and displays the results in the browser.

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

None

SEE ALSO

felix/surveyor/monitor

felix/surveyor/postgres

AUTHOR Joseph J. DesMarais

NOTES

```
 Format of GUI Request Pkt:
 ------------------------
    <Type Of Graph - int>
    <Type of Stat  - int>
    <Src Monitor   - String>
    <Dst. Monitor  - String>
    <StartTime     - String>
    <End Time      - String>
    <DB filename   - String>      allows use of simulated DB or actual DB
    <user id.      - String>
    <Parameter     - String(1 or more)>

    Parameter String:
     Topology then Parameter        ::=  <algorithm id>
     Prob Densty Func then Parameter::=  <# of bins>
     Autocorrelation then Parameter ::=  <max time lag>
     Log Log Distrbtn then Parameter::=  <# of bins>
     Variance Time                  ::=  <Max. block size>
     Stats Chart                    ::=  digit 0
     Time Series then Parameter     ::=  <subsmple factor><smooth blk size>
                                 <overlap factor>

    Graph Type ::= [0 - Time Series | 1 - PDF | 2 - Autocorrelation |
                3 - Log.Log.Distribution  | 4 - Topology |
                5 - Time Variance         | 6 - Stats Chart]

    Stat Type  ::= [0 - Delay | 1 - Loss | 2 - Throughput]
```

```
DB file entry format
------------------------
 <sender           - string> dotted decimal
 <receiver         - string> dotted decimal
 <timestamp        - string> ctime() format
 <end timestamp    - string> ctime() format
 <delay            - int>    mSec
 <pkts. Loss       - int>
 <bandwidth        - int>    kb/sec
 <pkts received    - int>
 <monitor version  - string>
```

A.2. corr

NAME

corr

SYNOPSIS

felix/corr/corr

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Produces a path file based upon analysis of the end-to-end delay times output by *fxsim*. This path file can be used as input to *mat*. The operation of *corr* is controlled by the parameters file "params.corr". "Params.corr" contains fields for setting the number of monitors, "n_mons", the simulation duration, "sim_duartion", and a noise threshold, "null_threshold".

PERMISSIONS: none

OPERANDS - none

USAGE - *corr*

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

"params.corr"

SEE ALSO

Fxsim, mat

AUTHOR Jason Baron

A.3. fan.cgi

NAME

*fan.cg*i – cgi-script that demonstrates the Distance Realization method of topology discovery based on form data from fan.html.

SYNOPSIS

felix/surveyor/cgi-bin/fan.cgi [options - none]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This cgi script is used to respond to forms generated by felix/surveyor/fan.html

PERMISSIONS: Permissions are based on the permissions of the executing web server. Write permission must be granted to the directory that the gif graph will reside. The default is /u/surveyor/public_html/pub

OPERANDS - none

USAGE - This program executes in the context of the web server.

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0	execution successful.
>0	An error occurred.

BUGS

None known

FILES

Temporary gif files are stored in /u/surveyor/public_html/pub/ on the web server.

SEE ALSO

felix/public_html/surveyor/cgi-bin/paul.cgi

felix/public_html/surveyor/cgi-bin/felix.cgi

AUTHOR Joseph J. DesMarais

A.4. felix.cgi

NAME

*Felixcg*i – cgi-script that queries an archive server and plots statistical results to be displayed in a web browser based on form data from felix.html.

SYNOPSIS

felix/surveyor/cgi-bin/felix.cgi [options - none]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This cgi script is used to respond to forms generated by felix/surveyor/felix.html

Upon reception of the query string a socket to the appropriate

archive server is opened, by this script, a tcp pkt with the below format is transmitted and the archive server returns a binary file containing a graph in gif format. This script then composes a page on the fly and presents the URL of gif graph and its context. The graph is stored temporarily on the web server machine then discarded as this program exits.

Format of Request Pkt:

```
-----------------------
<Pkt size      - int>          in bytes, not counting these 4 bytes       <Type Of Graph - int>
<Type of Stat  - int>
<Src Monitor   - String>
<Dst. Monitor  - String>
<StartTime     - String>
<End Time      - String>
<DB filename   - String>       allows use of simulated DB or actual DB
<user id.      - String>
<Parameter     - String(1 or more)>
```

Parameter String:

```
Topology then Parameter        ::=  <algorithm id>
Prob Densty Func then Parameter::=  <# of bins>
Autocorrelation then Parameter ::=  <max time lag>
Log Log Distrbtn then Parameter::=  <# of bins>
Variance Time                  ::=  <Max. block size>
Time Series then Parameter     ::=  <subsmple factor><smooth blk size
                                                     <overlap factor>
```

Graph Type ::= [0 - Time Series | 1 - PDF | 2 - Autocorrelation | 3 - Log.Log.Distribution | 4 - Topology | 5 - Time Variance | 6 - Stats Chart]

Stat Type ::= [0 - Delay | 1 - Loss | 2 - Throughput]

Permissions: Permissions are based on the permissions of the executing web server. Write permission must be granted to the directory that the gif graph will reside. The default is /u/surveyor/public_html/pub

OPERANDS - none

USAGE - This program executes in the context of the web server.

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

Temporary gif file is stored in /u/surveyor/public_html/pub/tmp.gif on the web server.

SEE ALSO

felix/public_html/surveyor/cgi-bin/fan.cgi

felix/public_html/surveyor/cgi-bin/paul.cgi

AUTHOR Joseph J. DesMarais

A.5. fixdrift

NAME

fixdrift – removes clock drift from opposite direction paths.

SYNOPSIS

```
felix/fixdrift/fixdrift [-o <outfile>] [-h] [-t] [-a] [-e] [-m] [-w <windowsize>] <infile1> <infile2>
```

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

fixdrift removes clock drift effects from two X-Y files that contain delay measurements (over the same time period) in opposite directions between the same two endpoints (for example, the path from monitor A to monitor B and the path from monitor B to monitor A). The average of the lower envelopes of the delay curves (one inverted) is assumed to be a good approximation of the clock drift effect. This average is added to or subtracted from the X-Y files to remove the effect. The output of the program is two adjusted X-Y files. They have the same base names as the input files and the extension ".cd#", where # is the number for the method used to combine the envelopes. Intermediate files can also be generated. If envelope files are generated, they have the letter e appended to the names of the output files. The envelopes with their means shifted to zero have the letters eA appended to the names of the output files. The combined envelope file has the name of the first output file with the letter E appended.

OPERANDS

-o <outfile>	specifies the output file for program messages. Defaults to stdout.
-h	print program description and usage information.
-t	enables printing of program trace information.
-c <method>	specify method for combining of two envelope functions. 1 = averaging. 2 = averaging for values of same sign; zero for values of opposite sign. 3 = combining to minimize change; average for values of opposite sign. 4 = combining to minimize change; zero for values of opposite sign (default).
-e	generate envelope files.
-M	use median value to shift envelopes instead of mean.
-T <trim>	specify fraction of data to trim off high and low ends (of pdf) when computing means for shifting envelopes.
-w <window>	specify window size (in seconds) for filtering. Defaults to 300 seconds.
<infile1>	first X-Y input file.
<infile2>	second X-Y input file. Contains delay measurements in opposite direction from <infile1>.

EXAMPLES

```
fixdrift 128096073102_192004018061.dat 192004018061_128096073102.dat
```

creates drift-corrected X-Y files 128096073102_192004018061.cd4 and 192004018061_128096073102.cd4. The window size used in computing the envelope curves is 5 minutes (300 seconds). The envelope curves are combined to minimize change: for each time interval, for envelope values of the same sign, the envelope value with the smallest absolute value is used as the combined value; for values of the opposite sign, zero is used as the combined value. Each envelope curve is shifted by its mean value before the curves are combined.

```
fixdrift –c 1 –T 0.05 -e A_B.dat B_A.dat
```

creates drift-corrected X-Y files A_B.cd1 and B_A.cd1. The window size used in computing the envelope curves is 5 minutes (300 seconds). The envelope curves are combined by averaging the values for each curve in each time interval. Each envelope curve is shifted by its mean value before the curves are combined. The lowest 5% of the values and the highest 5% of the values for each X-Y file are discarded before the envelope means are computed. Files containing the envelope curves are generated: A_B.cd1e, B_A.cd1e, A_B.cd1eA, B_A.cd1eA, and A_B.cd1E.

ENVIRONMENT VARIABLES

FELIX	specifies the path for the home directory of the Felix package.
HOSTMACH	specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

<infile1>	File containing list of monitor names or addresses, separated by newlines.
<infile2>	Path file generated by the program. Should have the extension .pth.

SEE ALSO

fxplot(1)

AUTHOR Bruce S. Siegell

NOTES

fxplot can be used to display the input and output files.

A.6. fixed

NAME

fixed

SYNOPSIS

```
felix/fixed/fixed filename
```

DESCRIPTION

Utility used to help fix the location monitors with the topomap
program. The single argument is a topomap generated file that contains
monitor locations. Output is written to standard out.

EXAMPLES

Example 1: To output to the screen the monitor locations, try

```
fixed gif1_mlabels
```

Example 2: To output monitor locations to a file, try

```
fixed gif1_mlabels > f
```

AUTHOR Jason Baron

A.7. fxplot

NAME

fxplot – constructs graphs showing various statistics of monitor data traces.

SYNOPSIS

felix/fxplot/fxplot [-f <dbfile>] [-o <graphfile>] [-e <filetype>] [-p] [-x <xyfile>] [-g <graphtype>]

[-all][-s <source> -d <dest>] [-M <mapfile>] [-E <extension>][-cc <ccrfile>][-c1 <ccrmon1> -c2 <ccrmon2> -c3 <ccrmon3> -c4 <ccrmon4>][-TS \"<starttime>\"] [-TE \"<endtime>\"] [-ts <starttime>] [-n <numpoints>][-c <dist>] [-r <maxrange>][-T <title>] [-l <label>]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

fxplot is a specialized pre-processing engine for graphing and assessing statistical characteristics of delay measurement datasets. The input file format can be Felix ascii database file, X-Y data file, or .ccr file (output of regress program). fxplot produces the following statistical graphs (though some may be idiosyncratically implemented):

0	time series
1	probability density function, f(x)
2	autocorrelation
3	log-log probability distribution log(1-Fx)
7	cross-correlation
9	statistics summary
10	dual-plot (time series and cross-correlation)

A large number of parameters including source files, graph type, parameters particular to each graph type, output format, etc are specified in an ascii parameters file (params.felix), and some can be defined on the command line. The command line values override the file values, allowing the user to set up a default environment in the parameters file, and vary some specifics from plot to plot in the command line within a shell script. Typical parameters are the source/destination monitor identity, start and end time of analysis within a time series, bounds for variable ranges on graphs.

fxplot creates a command file and a data file for gnuplot, which it calls. It will optionally run xv to display the resulting GIF image. fxplot is designed for rapidly creating and viewing a large number of graphs in sequence, so the user can quickly assess the characteristics of a large set of measurement data.

OPERANDS

-f <filename>	sets data file name. File may be Felix ascii database format or Felix X-Y data format.
-o <filename>	sets output file name for graph. Extension will be appended.
-e <filetype>	sets file type and extension for graph file.
-p	enables generation of x-y file.
-x <filename>	sets output file name for x-y file. The output file saves the data from a database file in two column format.
-g <graphtype>	sets graph type (see below).
-all	causes all Source-Destination pairs to be plotted (for ascii database file only).
-s <source>	sets source monitor symbol.
-d <dest>	sets destination monitor symbol.
-M <mapfile>	sets file containing mappings between symbols and names/IP addresses.
-E <extension>	sets extension for x-y input files. Defaults to \"dat\".
-cc <ccrfile>	sets cross-correlation file for graph types 7 & 10.
-c1 <ccrmon1>	sets first monitor (source) for cross-correlation.
-c2 <ccrmon2>	sets second monitor (destination) for cross-correlation.
-c3 <ccrmon3>	sets third monitor (source) for cross-correlation.

-c4 <ccrmon4>	sets fourth monitor (destination) for cross-correlation.
-ts <starttime>	sets the time series start time (in seconds since Jan. 1, 1970).
-TS \"<month> <day> <year> <hour> <minute>\"	sets the time series start time.
-TE \"<month> <day> <year> <hour> <minute>\"	sets the time series end time.
-TX <maxx>	sets maximum x value for time series plot (hours).
-TY <maxy>	sets maximum y value for time series plot (msec).
-n <numpoints>	sets the time series maximum number of points.
-c <dist>	shows the specified probabality distribution with the same mean and standard deviation as the data. (Only with the pdf graph type.)
-r <maxrange>	sets maximum range for delay measurements.
-G	turn off display of the generated graph. (For batch operation.)
-T <title>	sets title for graph.
-l <label>	additional label for graph.

EXAMPLES

fxplot -s A -d B -TS "Jun 26 1998 18 0" -TE "Jul 2 1998 18 0"

– specifies source, destination monitors and start and ending times within timeseries

ENVIRONMENT VARIABLES None

EXIT STATUS Not set

BUGS Some of the graph types advertised are not implemented

FILES

params.felix – file of run-time parameters for fxplot

SEE ALSO

fxsim has a similar type of parameters file. fixdrift output can be viewed using fxplot. See additional comments in fxplot source code.

AUTHORS Mark W. Garrett, Bruce S. Siegell

A.8. fxsim

NAME

fxsim

SYNOPSIS

```
felix/fxsim2.0/fxsim [-top <file>][-e <time>][-st <time>][-i <time interval>]
          [-r <#repeats>][-ran {0|1}][-t <trace level>]
          [-q {0|1}][-?]
```

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

The Felix Simulator "fxsim" uses discrete event simulation techniques to simulate the Felix experiment of monitors exchanging packets across a network. It reads in a parameter file "fxsim.par". Some of these parameters can be overridden by command-line arguments. The network is described by a topology file, in Felix Network Topology File format.

Each monitor sends packets to the other monitors. Each interval between sending packets from a given source, and each destination monitor, is chosen randomly and independently. The routing is specified in the topology file. Packets suffer delay due to link transmission times, and due to queuing at intermediate nodes. Link transmission delay is based on the packet size and the link characteristics. Queuing delay is either generated according to a random distribution specified in the topology file, or generated by actually simulating a queue at each node.

Options

-e <time>	Sets the end time of the simulation. Any packets still in the system at this time are ignored.
-i <time interval>	Sets the average time between sending packets from any given monitor.
-q {0|1}	1/0 sets queuing on/off in nodes.
-r <#repeats>	Sets the number of times the experiment is run.
-ran {0|1}	1/0 sets random numbers on/off.
-st <time>	Sets the time to stop sending packets. The simulation will continue to run until all packets have either been dropped or reached their destinations.
-t <trace level>	Sets trace level.
-top <file>	Sets the topology file to <file>.

The parameter file must look like the following, allowing for comment lines that start with the two characters "/*", and may be inserted anywhere:

```
Parameters for fxsim:
topology file: <file>
generate database file: {0|1}
generate graph: {0|1}
graph type: <type number>
graph for all or single SD pair: {0|1}
source, destination: <source number>,<destination number>
link: <loss/load link number for plot type 6>
end of sim [sec]: <time>
mons to stop sending [sec] <time>
monitor pkt interval [sec]: <time interval>
repeat: <#repeats>
random numbers on: {0|1}
trace: <trace level>
trace for single stn: <strace>
node to trace: <node number>
queueing at nodes on: {0|1}
packet size [bytes]: <size>
buffer size [pkts]: <size>
voice model on: {0|1}
speakers: <number>
voice rate [bps]: <rate>
lambda: <voice model parameter lambda>
mu: <voice model parameter mu>
xmin: <min>
xmax: <max>
```

bin count: <count>

If the generate graph parameter is 1, and the graph type is 0, or if the generate database file parameter is 1, the parameter file should continue with the lines (after the bin count and a blank line):

start time [sec]: <time>
number of packets: <count>

Either an ASCII database file should be written, or one of several graphs is plotted, using gnuplot. The database file, if generated, describes for each packet, where and when it was sent, and where and when it was received, but nothing about the route along the way.

If a graph is generated, it has one of the following types:

0	time series
1	probability density function f(x)
2	autocorrelation (takes a long time!)
3	log-log probability distribution log(1-Fx)
4	N/A (topology in felix system)
5	delay-loss tradeoff curve (1-Fx + loss)
6	loss as function of load

The parameters xmin, xmax, and bin count specify how the plots are done for graph types 1, 3, and 5.

The voice model parameters allow for simulating IP telephony traffic, instead of the standard Felix experiment.

FILES

fxsim.par	input parameter file
fxsim-db0.db	output ASCII database file

BUGS

AUTHORS Mark W. Garrett , Alex Poylisher, Jason Baron

A.9. fxtree

NAME

fxtree – tries to fit a tree with asymmetric link delays to packet delay data

SYNOPSIS

```
felix/fxtree/fxtree  [-l <number of lines> ][-s <scale factor>][-u][-c][[-noforwards | -forwards]
            [-m |-d0|-d1|-d2]
```

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This program reads a delay matrix, or a file of packet delays, and tries to fit this information by a tree with monitors on the leaves, and asymmetrical link delays. The delay information is read in from standard input in a format specified by the options. The tree is written out to standard output in the standard Felix topology file format. Each edge of the tree corresponds to two link records – one in each direction.

If the link delays are not required to be nonnegative, there will be a linear space of equivalent solutions of dimension equal to the number of internal nodes, which is most likely the number of monitors minus two. If the link delays are required to be nonnegative, there may be multiple solutions, a single solution, or no solutions at all. The program will also run substantially slower, since it will try to solve nonnegative least squares problems, rather then normal least squares problems.

Options are:

-c tries to center the solution in the set of equivalent solutions. The default is not to bother.

-db0,-db1,-db2 interprets the input data to be in one of three versions of the Felix Ascii Database File, as generated by one of three versions of the Felix simulator . "-db1" is the default.

-forwards writes the forwarding table entries in the output topology file. This is the default.

-l <line count> only read in <line count> packet info lines, if in one of the Database formats.

-m interprets the input data to be a delay matrix, with possible header lines specifying number of monitors and monitor names.

-noforwards don't bother to write the forwarding table entries.

-s <scale factor> rescale the link delays by the scale factor. The default is 0.000001 when using one of the Database formats, and 1.0 when using the matrix format. This rescaling is the inverse of the rescaling performed by the Felix simulator, and so is required to properly compare the simulator input with the inferred tree.

-u don't require that the link delays be nonnegative. The default is to require nonnegativity.

EXAMPLES

Example 1: To read the delay matrix "delay", try to fit a tree with nonnegative link lengths, and write it out to "foo.top" without forward table lines.

```
example%  fxtree –m < delay > foo.top
```

Example 2: To read in the simulator output file "fxsim.db", try to fit a tree with possibly negative link lengths, and write it out to "bar.top" including forward table lines.

```
example% fxtree –d1 <fxsim.eb > bar.top
```

ENVIRONMENT VARIABLES

All are ignored.

EXIT STATUS

0 All information was written successfully.

>0 An error occurred.

BUGS

Solutions are rarely unique.

SEE ALSO

fxsim

AUTHOR David F. Shallcross

NOTES

Requires LAPACK, and therefore Fortran libraries, to compile

A.10. imm

NAME

imm

SYNOPSIS

felix/imm/imm

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Used to compare two graphs. Reports if the graphs are the same or different, including immersions or "split" nodes.

PERMISSIONS: none

OPERANDS – two topology files

USAGE - *imm* topologyfile1 topologyfile2

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

SEE ALSO

Topology file

AUTHOR Jason Baron

A.11. mat

NAME

mat – matroid method for topology discovery, produces graph from path-link list

SYNOPSIS

felix/mat/mat

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Uses graph theory and assumptions about the graph structure to produce a graph and corresponding topology file from a path-link matrix.

PERMISSIONS: none

OPERANDS – path file

USAGE - *mat* pathfile

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

Pathfile

SEE ALSO

Path file, topology file

AUTHOR Paul Seymour

NOTES

One can modify the assumptions that *mat* makes about the network by defining various parameters in the source mat.c file. For example, to turn on the assumption that monitors are leaf nodes in the network make sure to include the C directive: “#define MONITORLEAF”.

A.12. matroid

NAME

matroid

SYNOPSIS

```
felix/matroid/matroid <# nodes> <# connectivity> <#monitors>
```

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Creates an initial graph based upon the command line arguments. Reduces this graph using the program reduce. The resulting path-link matrix is given as an argument to *mat*. *Mat* then attempts to reproduce the original graph. The original and reproduced networks are then drawn to create two .gif files. Finally, the two topology files are compared using *imm*.

PERMISSIONS:

OPERANDS - #nodes, #connectivity, #monitors

USAGE -

EXAMPLES – matroid 20 5 5

ENVIRONMENT VARIABLES - none

EXIT STATUS

0	execution successful.
>0	An error occurred.

BUGS

None known

FILES

Produces the files gif1.gif, gif2.gif, and imm.output

SEE ALSO

mat, imm, reduce

AUTHOR Jason Baron

A.13. monsolaris

NAME

monSolaris – This program is an implementation of an active network probe for collecting one way, end to end packet delay and packet loss.

SYNOPSIS

```
felix/monitor/monSolaris [-a | -h | -d] [-p <port>] [-c <config file>] [-m <model>] [-s <server addr>] <init filename> <passwd>
```

-a	permits the writing of archive files
-h	allows web access & embeds HTML tags in archive

-d view debug statements

-p <port> local port number for inter-monitor communication

-c <config file> config file to read model parameters

-m <model> controls packet transmission rates: random, voice simulation

-s <server> dotted decimal address of the archive server

<init file> text file containing a list of at least one other monitors ip address and port number.

<passwd> used in MD5 authentication of the inter-monitor packets.

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This program uses UDP to receive and transmit network monitoring messages. The program act as both a transmitter and a receiver. A monitor transmitter sends a message to the monitor receiver. The receiver decodes the records of the sending station and updates them if it is a host that has had prior participation. Otherwise, it adds an entry for the new monitoring station. In addition to the sending time stamp and sequence number the monitor packets contain a listing of other monitor stations that a sender is aware of. This information is stored by the receiver and used to expand the receivers view of the network. In this sense the monitors learn of other monitors and knit themselves into a unified mesh.

The transmit messages are sent at randomly chosen time intervals to a station with which it has had prior knowledge. The choice of destination station is also made at random. The monitors calculate one way delays by taking the difference of sending timestamp and the timestamp the packet is received. Packet loss is detected via comparing a packets embedded current sequence number with the sequence number of the last packet received from that host. Gaps indicate lost packets. The delay and loss calculations are sent to an archive server which stores them in an SQL DBMS for later use by the analysis front end.

ENVIRONMENT VARIABLES - none

EXIT STATUS

0 execution successful.

>0 An error occurred.

BUGS

None known

FILES

<config file> is a text file containing IP addresses and port numbers of other monitor hosts.

SEE ALSO

felix/surveyor/dbPostgres/archsvr

AUTHOR Joseph J. DesMarais

NOTES

```
------------------------------------------------------------------
                  Message Byte Layout
----------------------------------------------------------------
bytes type    field       buffer byte positions  semantics
----------------------------------------------------------------
```

4	u_long	rIpAddr;	0-3	not sent remote station's address
2	u_short	rPort;	4-5	not sent remote station's port
4	u_long	lIpAddr;	6-9	not sent local station's address
2	u_short	lPort;	10-11	not sent local station's port
4	int	msgType;	12-15	software version
1*16	char	md5[16];	16-31	signature
4	int	incarn;	32-35	ids the instance
4	int	seqNum;	36-39	ids source dest pair
4*2	int	sendTime[2];	40-47	sending date & time local clk
4	int	peerIncarn;	48-51	last incarn
4	int	peerSeqNum;	52-55	last serial
4*2	int	lastWhen[2];	56-63	date & time of last message
4*2	int	rcvWhen[2];	64-71	local value of last rcvd mesg

----

72 <=== Sub Total Bytes ===> 72
Overhead

Per station information

4	u_long	nIpAddr;	72-75	IpAddr of a network station
2	u_short	nPort;	76-77	port of a network station
4	int	lossRate;	78-81	
4	int	delay;	82-85	in milliseconds
4	int	statTime**;	86-89	time stats were calculated S
4	int	throughput;	90-93	zeroed in this version

----

22 <=== Sub Total Host Data Set Bytes

----

94 <===== Total Bytes =====> 94 Minimum size of a status message.
assuming 1 per station
data set

----

general case byte count:

72 + (N * 22) bytes.

Where N is the number of host data sets in the row being sent.

1492 - 72 = 1420; 1420 / 22 = 64.54 Per Station Data sets Max

Function Calls:

```
main()
  sendMessage()
    compose_message()
    encodeMD5()
  receiveMessage()
    decodeMD5()
    parseMessage()
    addNewHost()
```

```
        notFound()
        diff_timeval()
    update()
        notFound()
        update_lossRate()
        diff_timeval()
        update_delay()
            millisec()
            diff_timeval()
    writeRaw()
        millisec()
writeArchive()
    millisec()
dump()
    millisec()
loadWebPage()
    millisec()
    diff_timeval()
```

A.14. pathconvert

NAME

pathconvert – convert a path file into various other formats

SYNOPSIS

felix/pathconvert/pathconvert –b <basename> [-e <errfile>] [-o <outfile>] [-t]
[-a] [-B] [-c] [-j] [-p] [-P] [-n] [-s] [-S] [-x] [-y] [-z]

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

pathconvert generates various types of files from path information collected using traceroute. The format of the input file is that generated by the tracepaths program. Each line of the input file contains the information for one path, e.g.,

buzzard sherry : 128.96.73.102 128l.96.215.253 128.96.210.137

If no input file is specified, then input is taken from stdin. Input files should not be preceded by a '-' character. By default, program messages are printed to stdout, and error messages are printed to stderr. The messages can be redirected to files using the –o and –e options respectively.

OPERANDS

-b <basename>   specifies the base file name for output files.

-e <errfile>   specifies the error message file. Defaults to stderr.

-o <outfile>   specifies the output file for regular program messages. Defaults to
stdout.

-h   prints program description and usage information.

-t   enables printing of trace information.

-a   creates all output files. Equivalent to –c –j –p –P –n –s –l.

-B   causes links to be treated as bidirectional By default, links are assumed to be unidirectional.

-c   creates files identifying paths that share common components. The
output will be in <basename.>.common and <basename>.com2.

-C   creates a file with the count of the number of paths each node appears in.
The output will be in <basename>.ncnt

-j	creates an html file with net list parameters for the Graph java applet.
The output will be in <basename>_java.html

-l	creates a links on paths file listing the symbols for the endpoints of each
path (e.g., A, B, etc.) and symbols for the links that the path crosses (e.g., l001, l002, …). The output will be in <basename>.lop

-p	creates a paths crossing links file. The output will be in <basename>.lpth.

-P	creates a reduced paths crossing links file. The output will be in <basename>.lpthr.

-n	creates a Felix Topology File and a simple net list file. The outputs will be in <basename>.top and <basename.net> respectively.

-s	creates a copy of the path file with symbols instead of names and addresses. The output will be in <basename>.spth.

-S	disables creation of a symbol mapping file. By default, the symbol file is created. The output will be in <basename>.spth.

-u <symfile>	specifies the symbol file (.sym) to use as input for the program. This file is used to map the node names and addresses in the path file to shorter symbol names. The symbol file may also contain the locations of some or all of the nodes.

-x	merges nodes using DNS information before generating files.

-y	merges nodes that enter other nodes at the same interface before generating files.

-z	merges serial edges before generating files.

<input>	specifies the path file to use as input. Defaults to stdin. The input file is typically called <basename>.pth, but the full name must be specified explicitly.

EXAMPLES

pathconvert –b surveyor surveyor.pth

generates a symbol mapping file (surveyor.sym) with short symbols for each of the nodes referenced in the path file. Leaf nodes (assumed to be monitors) are assigned symbols from A-Z in the order that the nodes are encountered in the path file. Internal nodes are assigned symbols n###. The generated symbol file can be modified (by hand) to add location information or to give different symbol names to the nodes.

pathconvert –u surveyor.sym –b surveyor -a surveyor.pth

converts the path information in surveyor.pth into all of the file formats that pathconvert can generate. surveyor.sym specifies the names to use for nodes listed in surveyor.pth and may specify the locations of some of the nodes.

pathconvert –u surveyor.sym –b surveyor –n –S surveyor.pth

generates a Felix Topology File (surveyor.top) and a simple net list file (surveyor.net) describing the connectivity of the network described by the paths. Either of the output files can be used as an input file for topomap for rendering a graph of the topology.

ENVIRONMENT VARIABLES

FELIX	specifies the path for the home directory of the Felix package.
HOSTMACH	specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

xxx.pth	path file; input file for the program

xxx.sym	symbol mapping file
xxx.top	Felix Topology File
xxx.net	simple netlist file

SEE ALSO

tracepath(1), topomap(1)

AUTHOR Bruce S. Siegell

A.15. pathfilter

NAME

pathfilter – selects a subset of paths from a path file.

SYNOPSIS

felix/pathfilter/pathfilter [-e <errfile>] [-o <outfile>] [-h] [-t] [-s <monfile> | -S "<mon-list>"] [-d <monfile> | -D "<mon-list>"] [-m <monfile> | -M "<mon-list>"] [<infile>]

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Pathfilter generates a path file that contains a subset of the paths in the given path file. The format of the input and output files is that generated by the tracepaths program. Each line of the input/output file contains the information for one path, e.g.,

buzzard sherry : 128.96.73.102 128l.96.215.253 128.96.210.137

If no input file is specified, then input is taken from stdin. Input files should not be preceeded by a '-' character. By default, output is written to stdout, and error messages are printed to stderr. The messages can be redirected to files using the –o and –e options respectively.

OPERANDS

-e <errfile>	specifies the error message file. Defaults to stderr.
-o <outfile>	specifies the output file for regular program messages. Defaults to stdout.
-h	prints program description and usage information.
-t	enables printing of trace information.
-s <monfile>	specifies the name of a file containing a list of source monitors to be included in the generated path file.
-S "<mon-list>"	provides a list of source monitors (separated by spaces) to be included in the generated path file.
-d <monfile>	specifies the name of a file containing a list of destination monitors to be included in the generated path file.
-D "<mon-list>"	provides a list of destination monitors (separated by spaces) to be included in the generated path file.
-m <monfile>	specifies the name of a file containing a list of monitors to be included in the generated path file. May be used in place of –s, -d, or both.
-M "<mon-list>"	provides a list of monitors (separated by spaces) to be included in the generated path file. May be used in place of –S, -D, or both.
<input>	specifies the path file to use as input. Defaults to stdin.

EXAMPLES

pathfilter –s sources.txt –d destinations.txt –o newpaths.pth surveyor.pth

generates a path file listing only the paths from surveyor.pth that have the source names specified in sources.txt and the destination names specified in destinations.txt. The output is written to newpaths.pth.

pathfilter –S cmu.csg –d monitors.txt –o cmuout.pth surveyor.pth

generates a path file listing only the paths starting from the node named cmu.csg and with destination names specified in monitors.txt. The output is written to cmuout.pth.

pathfilter –S cmu.csg –D "harvard.csg arl.dren virginia.csg" –o cmuout3.pth surveyor.pth

generates a path file listing only the paths starting from cmu.csg and ending at harvard.csg, arl.dren, or virginia.csg. The output is written to cmuout3.pth.

pathfilter –M "cmu.csg harvard.csg arl.dren virginia.csg" surveyor.pth

generates a path file listing all the paths between the 4 specified nodes. The output is written to stdout.

ENVIRONMENT VARIABLES

FELIX specifies the path for the home directory of the Felix package.

HOSTMACH specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

xxx.pth	path file; input file for the program
xxx.txt	monitor file, containing the name of one monitor per line.

SEE ALSO

tracepath(1), pathconvert(1)

AUTHOR Bruce S. Siegell

NOTES

The names specified in the monitor files or the –S, -D, and –M options must match the names used for the sources and destinations in the path file provided as input. There is no symbol or address lookup in this program.

A.16. paul.cgi

NAME

*paul.cg*i – cgi-script that demonstrates the Matriod method of topology discovery based on form data from paul.html.

SYNOPSIS

felix/surveyor/cgi-bin/paul.cgi [options - none]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This cgi script is used to respond to forms generated by felix/surveyor/paul.html

Upon reception/parsing of the query string one of 3 programs are called

paul.c, save.c or label.c

paul.c:

compose georgia tech random graph

run pauls algorithm

compose resulting graphs: orig topology and candidate
topology (gif)

args: nodes (1-3), complexity (1-3), monitors (1-3)

save.c

moves files assoc with a paul.c run to some destination
directory

args: path (string)

nominally starting at ~surveyor/public_html/pub

NOTE: public_html portion of the path is implied by the browser

lable.c

adds or removes node or monitor labels on the output gif

args: nodes(1-on, 0-off), monitors (1-on, 0-off)

This program displays the results of the above actions by composing an

html page on the fly for presentation. The matroid, re-labing and saving can be executed from the same form instance in a serial format.

viewing the archive short circuits any choices that may be contanined in form instance.

This program also displays the contents of the archive directory via the form retreive command and presents links to view the files.

PERMISSIONS: Permissions are based on the permissions of the executing web server. Write permission must be granted to the directory that the gif graph will reside. The default is /u/surveyor/public_html/pub

OPERANDS - none

USAGE - This program executes in the context of the web server.

EXAMPLES - none

ENVIRONMENT VARIABLES - none

EXIT STATUS

0	execution successful.
>0	An error occurred.

BUGS

None known

FILES

Temporary gif files are stored in /u/surveyor/public_html/pub/<time of day>.gif and /u/surveyor/public_html/pub/<time of day + 1>.gif on the web server.

SEE ALSO

felix/public_html/surveyor/cgi-bin/fan.cgi felix/public_html/surveyor/cgi-bin/felix.cgi

AUTHOR Joseph J. DesMarais

A.17. pgconvert

NAME

pgconvert – generates X-Y files from information extracted from a Postgres database.

SYNOPSIS

```
felix/pgconvert/pgconvert [-e <errfile>] [-o <outfile>] [-h] [-t]
                [-H <dbhost>] [-s <source> -d <dest>] [-p <paramater>]
                [-S "<starttime>"] [-E  "<endtime>"]
                <infile>
```

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Pgconvert generates X-Y files from information extracted from a Postgres database. It can generate either a single X-Y file for a specified pair of monitors, or all X-Y files for a given list of monitors. The database name is assumed to be "felix" on the specified host.

OPERANDS

-e <errfile>	specifies error message file. Defaults to stderr.
-o <outfile>	specifies output file. Defaults to stdout
-h	print program description and usage information.
-t	toggles printing of program trace information.
-H <dbhost>	specifies name of database server host.
-s <source>	specifies source host for extraction of single x-y file. Accepts either host name or IP address.
-d <dest>	specifies destination host for extraction of single x-y file. Accepts either host name or IP address.
-p <parameter>	specifies parameter to extract. Can be "delay", "pktloss" (packet loss), "pktsrcv" (packets received), or "bw" (bandwidth). Default value is "delay".
-S "<month> <day> <year> <hour> <minute>"	specifies the start time. Use four digit year.
-E "<month> <day> <year> <hour> <minute>"	specifies the end time. Use four digit year.

<infile> specifies input file containing list of monitors (one per line). Defaults to stdin. Ignored if -s and -d are specified. Monitors can be specified as either host names or IP addresses.

EXAMPLES

```
pgconvert -H sundown -s 128.96.73.102 -d 192.4.18.61 -p delay \
        -S "Dec 4 1998 0 0" -E "Dec 5 1998 0 0"
```

creates an X-Y file containing delay measurements between buzzard and brooklyn from 12:00am on December 4 until 12:00am on December 5 using information from the Postgres database on sundown. The created file will have the name 128096073102_192004018061.dat.

```
pgconvert -H sundown -p loss monitors.txt
```

creates X-Y files containing loss measurements between all monitor pairs (in both directions) from time 0 (Jan. 1, 1970) until the time the extraction is started. monitors.txt lists the names or IP addresses of the monitors.

ENVIRONMENT VARIABLES

FELIX specifies the path for the home directory of the Felix package.

HOSTMACH specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

<infile> File containing list of monitor names or addresses, separated by newlines.

AUTHOR Bruce S. Siegell

A.18. plotall.sh

NAME

p*lotall.sh* – converts surveyor binary formatted times stamp and delay data to felix XY standard file format.

SYNOPSIS

felix/plotall/plotall.sh <date 3CharMonth_day_year> <starting time of the data in seconds since 1970>

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Plotall.sh is a shell script that coordinates the activities of the following programs, unpackbin.sh, adxy2felixxy, truncate and fxplot. Unpackbin.sh converts the binary data files into ascii text, the data files are concatenated into one large data file. Adxy2felixxy converts the surveyor format into felix standard X, Y format. Truncate then converts the floating point delay values into integer milliseconds. At this point we have a standard felix XY file type. The XY file is then processed by fxplot and 3 gif graphs are rendered. The graphs are the time series, probability density, and log by log distribution of the delay data.

ENVIRONMENT VARIABLES - none

EXIT STATUS

0	execution successful.
>0	An error occurred.

BUGS

None known

FILES

None

SEE ALSO

AUTHOR Joseph J. DesMarais

A.19. process_mat

NAME

process_mat – transform a delay matrix to be acceptable to realize or weak.

SYNOPSIS

felix/realizations/process_mat [<input delay matrix file name>[<output delay matrix file name]]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This is a filter to read in a delay matrix, and transform it to be symmetric (by averaging it with its transpose), nonnegative (by maximizing with 0), and obedient to the triangle inequality (by running a shortest path algorithm). The transformed delay matrix is written out.

OPERANDS

The first argument, if present, is the name of the input delay matrix file. If there are no arguments, the input is read from standard input. The second argument, if present, is the name of the output delay matrix file. If there are fewer than two arguments, the output is written to standard output. If you want to read from standard input, but write to a file, you must use whatever output redirection features your shell has.

EXAMPLES

Example 1: To read in a matrix file "delay.mat" and writing a transformed matrix to standard output.

```
example%  process_mat  delay.mat
```

ENVIRONMENT VARIABLES

All are ignored.

EXIT STATUS

All information was written successfully.

>0	An error occurred reading or writing files.

BUGS

None known.

SEE ALSO

realize, weak

AUTHOR David F. Shallcross

A.20. real

NAME

real

SYNOPSIS

```
felix/real/real <# nodes> <# connectivity> <#monitors>
```

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Creates an initial graph based upon the command line arguments. Reduces this graph using the program reduce. The resulting path-delay matrix is given as an argument to *realize*. *Realize* then attempts to reproduce the original graph. The original and reproduced networks are then drawn to create two .gif files.

PERMISSIONS:

OPERANDS - #nodes, #connectivity, #monitors

USAGE -

EXAMPLES – matroid 20 5 5

ENVIRONMENT VARIABLES - none

EXIT STATUS

0	execution successful.
>0	An error occurred.

BUGS

None known

FILES

Produces the files gif1.gif and gif2.gif

SEE ALSO

realize, reduce

AUTHOR Jason Baron

A.21. realize

NAME

realize – create an undirected graph with a specified distance matrix between monitors.

SYNOPSIS

felix/realizations/realize [-clip <clip threshold>][-noforwards | -forwards][<matrix file> | -]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This program reads a symmetric, nonnegative delay matrix, and tries to fit this information by a graph with monitors among the nodes, and symmetrical link delays. The delay matrix is assumed to obey the triangle inequality: d(i,j) <= d(i,k) + d(k,j). Viewing the delays as arc lengths, the shortest paths between monitors will have total lengths equal to the given matrix values. The delay information is read in from a file or standard input in delay matrix format. The tree is written out to standard output in the standard Felix topology file format, with two link records with identical delays for each edge of the graph.

There are very many graphs which realize any given delay matrix. This program uses local search to try to find a graph with as small a total delay as possible. It is not expected to find the minimum total delay graph. The minimum total delay graph is very sensitive to the data – changing a matrix entry by a small amount may require adding new edges. To deal with this problem, numbers differing by a hard-coded epsilon will be treated as equal. The user also has an option of setting a clipping threshold, such that edges shorter than the threshold will be contracted.

Options are:

-clip <threshold>	edges shorter than <threshold> will be contracted. Default is zero.
-forwards	writes the forwarding table entries in the output topology file. This is the default.
-noforwards	don't bother to write the forwarding table entries.

OPERANDS

Either a delay file name or "-" may be specified. If a file name is specified, that file will be read. If "-" is specified, or no file name is given, then the delay matrix will be read from standard input.

EXAMPLES

Example 1: To read in a matrix file from standard input, and write a topology file for a realization to standard output.

```
example%  realize
```

Example 2: To read in a matrix file "delay.mat", and write out to standard output a topology file for a realization, with all links of delay less than 0.001 contracted.

```
example% realize –clip 0.001 delay.mat
```

ENVIRONMENT VARIABLES

All are ignored.

EXIT STATUS

0 All information was written successfully.

>0 An error occurred.

BUGS

Does not in general find minimum length graph. Real internets don't use minimum total delay routing.

SEE ALSO

fxtree, weak, process_mat

AUTHOR David F. Shallcross

A.22. reduce

NAME

reduce – adds monitors to a graph generated by tiers; creates internet-style routing between all pairs of monitors; "reduces" graph by deleting undiscoverable links and nodes (given the set of monitors)

SYNOPSIS

felix/reduce/reduce [-m [# of monitors]] [-p [hop | short]] [-o [output directory]] filename

DESCRIPTION

Chooses a number of monitors and places them on the leaves of the graph specified by filename. Shortest paths based upon end-to-end delay time or hop count determine the routing in the graph. The original graph is reduced by including only those edges that are traversed and by removing series edges.

Output includes the files: delay, delay2, path2 and topology.ns. "Delay" and "delay2" contain a matrix of the end- to-end delay times between all pairs of monitors before and after

series edges are removed, respectively. "Path2" contains the set of links between all monitor pairs in the reduced graph that does not contain series edges. Finally, "topology.ns" contains the structure of the reduced graph without series edges in topology file format.

OPTIONS

-m [# of monitors]

Specifies the number of monitors to be randomly placed on
leaves of the graph. If more monitors are specified then there are
leaves in the graph, then the program exits with an error
message. If the -o option is not specified, five monitors are
chosen by default.

-p [hop | short]

Specifies the desired routing to used. "Short" uses
paths that are of least delay between monitors, while "hop"
uses paths that contain the least number of hops
between monitors.

-o [output directory]

Specifies the directory where the output files should be placed.

EXAMPLES

Example 1: To run shortest hop based routing with ten monitors, try

```
reduce -m 10 -p hop filename
```

Example 2: To run shortest delay path routing with fifteen monitors, try

```
reduce -m 15 -p short filename
```

Example 3: To create a sample input file and run hop based routing, try

```
tiers 1 8 3 10 20 3 3 2 1 1 1 > filename; reduce -p hop filename
```

AUTHOR Jason Baron

A.23. regress

NAME

regress – performs correlation analysis of two time series (X-Y) files.

SYNOPSIS

```
felix/regress/regress -o <outfile> [-e <extension>] [-h] [-?] [-t] [-d] [-x]
        [-a <adjustment>] [-c <method>] [-f <function>]
        [-N <first>] [-n <points>] [-T <trim>] [-U <truncate>]
        [-S “<starttime>”] [-E “<endtime>”] [-i <interval>]
        [-w <windowsize>] [-w <windowstep>]
        [-s <src1> <dst1> <src2> <dst2>] [-M <mapfile>]
        <infile1> <infile2>
```

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

regress does correlation analysis of two X-Y data files (<infile1> and <infile2>), where the X values in both files represent times with the same base. Y values in the two files are paired for the correlation based on their X values. Prepared values contained in a single file (<infile1>) can also be correlated using the -x option.

OPERANDS

-o <basename> specifies the base name for output files.

-O <extension> specify output file extension - "gif" (GIF), "ps" (Postscript), or "ccr" (X- Y data). Defaults to "gif".

-e <extension> sets extension for x-y input files. Defaults to "dat".

-h or -? print program description and usage information.

-t enables printing of program trace information.

-d turn off display of graph on the screen. xv is used to display gif files. ghostview is used to display postscript.

-x read both x and y from a single x-y file.

-a <adjustment> adjust time boundaries. Round minimum down to nearest multiple of adjustment (in seconds) and maximum up to nearest multiple of adjustment. Defaults to 1 second. To disable, use adjustment of zero.

-c <method> specify correlation method.

1 = linear regression (default)

2 = cross correlation (implies functions 2 or 3)

3 = lambda correlation (implies functions 2 or 3)

-C <threshold> specify congestion threshold for lambda correlation.

-f <function> specify the type of output to generate.

1 = linear regression line and scatterplot

2 = correlation coefficients (r) for sliding window (default)

3 = coefficients of determination (r*r) for sliding window

4 = slopes of regression lines for sliding window

5 = t-scores for correlation coefficients for sliding window

6 = correlation coefficients (r) for sliding window with t score of at least 2.0

7 = slopes of regression lines for sliding window with t score of at least 2.0

8 = intercepts of regression lines for sliding window

-N <first> specify first point to process (ignore points before first point).

-n <points> specify number of points to process.

-T <trim> specify percentage to trim off each side (of pdf) to eliminate outliers.

-U <truncate> specify percentage to truncate off of one side (of pdf) to eliminate outliers. Positive for right side, negative for left side.

-S "<month> <day> <year> <hour> <minute>" sets the time series start time.

-E "<month> <day> <year> <hour> <minute>" sets the time series end time.

-i <interval> specify amount of time in seconds that each datapair represents. Defaults to 5 minutes (300 seconds).

-I <value> specify input value to ignore.

-w <windowsize> specify the size of the sliding window (in seconds). Defaults to 4 hours (14400 seconds).

-W <windowstep> specify step size between adjacent windows (in seconds). Defaults to windowsize/2.

-s <src1> <dst1> <src2> <dst2>

Specifies path pairs using the symbols representing the monitors. The src1 and dst1 symbols are translated into a file name for the first input file. The src2 and dst2 symbols are translated into a file name for the second input file. This option is used in place of <infile1> and <infile2>.

-M <mapfile> Specifies the file used to map between symbols, host names, and IP addresses for the monitors.

<infile1> first input file. Information used for independent variable. Not used with -s option.

<infile2> second input file. Information used for dependent variable. Not used with -s option or -x option.

EXAMPLES

regress –S "May 24 1999 16 0" –E "May 25 1999 0 0" –i 5 –w 300 –s cmu wisc harvard arl –M surveyor.sym –o cwha –O ccr

creates an X-Y file called cwha..ccr containing the correlation coefficients for 5 second samples at 5 minute (300 second) intervals (overlapping at 2.5 minutes) starting from 4:00pm on May 24, 1999 and ending at midnight (12:00am on May 25, 1999). The path from cmu.csg to wisc.csg and the path from harvard.csg to arl.dren are correlated. The symbol mapping file surveyor.sym is used to map the monitor symbols (e.g., cmu) to the monitor names (e.g., cmu.csg). The correlation method used is the one based on linear regression (almost identical to the cross correlation method, but requiring no interpolation).

ENVIRONMENT VARIABLES

FELIX specifies the path for the home directory of the Felix package.

HOSTMACH specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

<infile1>	File containing list of monitor names or addresses, separated by newlines.
<infile2>	Path file generated by the program. Should have the extension .pth.

SEE ALSO

gnuplot(1), xv (1), ghostview(1), fxplot(1)

AUTHORS Bruce S. Siegell

NOTES

gnuplot, xv, and ghostview must be in the PATH.

A.24. tiers

NAME

tiers - creates a realistic internet graph

DESCRIPTION

Command line parameters passed to tiers control the structure of the generated graph. The graphs are based upon a hierarchical three "tier" model, consisting of a WAN, MANs, and LANs. The number of nodes in each tier, the relative number of one type of tier to another tier, the connectedness within a tier, and the connectedness between tiers are all controllable through parameters.

SYNOPSIS

```
felix/tiers/src/tiers NW NM NL SW SM SL RW RM RL RMW RLM
```

NW: maximum number of WANs (currently only 1 supported)

NM: maximum number of MANs per WAN

NL: maximum number of LANs per MAN

SW: maximum number of nodes per WAN

SM: maximum number of nodes per MAN

SL: maximum number of nodes per LAN

Optional parameters:

RW: intranetwork redundancy for WAN (default 1)

RM: intranetwork redundancy for MANs (default 1)

RL: intranetwork redundancy for LANs (currently only 1 supported)

RMW: internetwork redundancy for MAN to WAN (default 1)

RLM: internetwork redundancy for LAN to MAN (default 1)

EXAMPLES

Example 1: To see just a WAN with some redundancy, try

```
tiers 1 0 0 30 0 0 3 1 1 1 1
```

Example 2: To see a WAN with some MANs, try

```
tiers 1 3 0 20 10 0 3 2 1 1 1
```

Example 3: To see a three tier network with redundancy, try

```
tiers 1 10 5 10 10 5 2 1 1 2 1
```

Example 4: to visualize the tier network using gnuplot, try

```
tiers 1 8 3 10 20 3 3 2 1 1 1 > myfile.gnu ; gnuplot "myfile.gnu"
```

AUTHORS K. Calvert and E. Zegura, Georgia Institute of Technology. This code is available at www.cc.gatech.edu/projects/gtitm [CAL97].

A.25. topomap

NAME

topomap – creates a graph from a net list file.

SYNOPSIS

```
felix/topomap/topomap –b <basename> [-c <count>] [-e] [-f] [-F]
[-h <height>] [-i] [-l <length>] [-n] [-N]
[-o <outfile>] [-t] [-w <width>]
[<infile>]
```

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

topomap creates gnuplot command and data files for rendering a graph of the given network list information. It executes the gnuplot file command file to generate a GIF file with the graph. Topomap's input is either a simple net list file (if the –i option is given) or a Felix Topology File.

If no input file is specified, then input is taken from stdin. Input files should not be preceeded by a '-' character. By default, program messages are written to stdout, and error messages are written to stderr. The messages can be redirected to files using the –o and –e options respectively.

OPERANDS

-b <basename> specifies the base file name for output files. The gnuplot command file will be called <basename>, and data files will be called <basename>_nodes, <basename>_edges, and <basename>_labels.

-c <count> specified the number of relaxation steps between checkpoints.

-e show edge labels.

-f fix marked nodes at particular locations. The locations are placed in the gnuplot space (a 1000 by 1000 grid).

-F fix marked nodes and scale their positions so that they fill the page. The locations are scaled to the gnuplot space.

-h <height> specifies the initial height of the graph (in pixels).

-i specifies the use of the simple net list file format for the input file instead of the Felix Topology File format.

-l <length> specifies the desired length of the edges (in pixels).

-n show leaf node labels only.

-N do not show any labels.

-o <outfile> specifies the output file for regular program messages. Defaults to stdout.

-r <radius> specifies the radius of a node (in pixels). The radius will be the specified number minus 0.5.

-s <speed> specified the maximum amount nodes can move in each relaxation step. For graphs with cycles, high speed can result in oscillation. Defaults to 5 pixels.

-t enables printing of program trace information.

-w <width> specifies the initial width of the graph (in pixels).

<input> specifies the path file to use as input. Defaults to stdin.

EXAMPLES

topomap –b surveyor –F surveyor.top

creates a GIF file (surveyor.gif) with a graph of the topology described in the Felix Topology File surveyor.top using the fixed locations of the nodes specified in the file.

ENVIRONMENT VARIABLES

FELIX	specifies the path for the home directory of the Felix package.
HOSTMACH	specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

xxx.top	path file; input file for the program
xxx.net	monitor file, containing the name of one monitor per line.
xxx	shell script generated by topomap. Calls gnuplot with the generated data files.
xxx_nodes	locations of nodes in the generated graph.
xxx_edges	description of line segments for edges in the generated graph.
xxx_labels	labels and their locations in the generated graph.

SEE ALSO

gnuplot(1), pathconvert(1)

AUTHOR   Bruce S. Siegell

A.26. tracepath

NAME

tracepath – generates a list of IP paths between the current host and a given set of hosts.

SYNOPSIS

```
felix/tracepath/tracepath [–h] [-n] [-t] [-e <errfile>] [-o <outfile>] [<infile>]
```

OPERATING SYSTEMS

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

Tracepath prints the paths taken by messages between the current host and the listed destination hosts. It uses traceroute to determine the paths. The input file is a list of host names or dotted decimal IP addresses with one name or address per line. If no input file is specified, then input is taken from stdin. Input files should not be preceeded by a '-' character. By default, the path information is written to stdout, and error messages are written to stderr. The messages can be redirected to files using the –o and –e options respectively.

The output of this command is a path file with each line containing the information about one path. For example, the line for the path from buzzard to sherry is

```
buzzard sherry : 128.96.73.102 128.96.215.253 128.96.210.137
```

OPERANDS

-h	print program description and usage information.

-n print host names instead of IP addresses for the path components.

-t enables printing of program trace information.

-e <errfile> specifies the error message file. Defaults to stderr.

-o <outfile> specifies the output file for the path information. Defaults to stdout.

<input> specifies the file to use as input. Defaults to stdin.

EXAMPLES

tracepath –o monitors.pth monitors.txt

creates a path file (monitors.pth) listing the paths from the current host to all hosts listed in monitors.txt.

ENVIRONMENT VARIABLES

FELIX specifies the path for the home directory of the Felix package.

HOSTMACH specifies the subdirectory within $FELIX/bin for installation of the binaries for the architecture under which the program was compiled.

EXIT STATUS

0	the program executed with no terminal errors.
>0	the program aborted with an error.

BUGS

Probably.

FILES

<infile> File containing list of monitor names or addresses, separated by newlines.

<outfile> Path file generated by the program. Should have the extension .pth.

SEE ALSO

traceroute(1), pathconvert(1)

AUTHOR Bruce S. Siegell

A.27. tree_merge

NAME

tree_merge – merge a set of rooted weak realizations

SYNOPSIS

felix/realizations/tree_merge [-p][-u] [<specification file>]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

This program merges a set of rooted weak distance matrix realizations into a strong realization. The set of rooted weak realizations is described by a specification file, which consists of a sequence of lines, each having the name of a monitor followed by the name of the topology file for the weak realization rooted at that monitor. Such a specification file would typically be written by the program "weak" with option "-rall". It is permissible to have weak rooted realizations for from one to all of the monitors. It probably won't work to have more than one realization with the same root. File names are interpreted as relative to the current working directory, rather than, say, the directory containing the specification file. If the specification file name is not given, the list of roots and file names is read from standard input.

From the rooted weak realizations a distance matrix is inferred, using the distances from the roots to the other monitors. The program looks for a graph of as small as possible total length, based on the rooted weak realizations, that is a strong realization for the inferred distance matrix. Depending upon the options, one of several methods may be chosen, and additional constraints may be imposed. The resulting graph is written to standard output as a topology file.

Options:

-p use a method based on merging paths that might be identical.

-u ensure that the resulting graph is "universal" for the input graphs. That is, that it contains subdivisions of each input graph as a subgraph.

OPERANDS

The following operand is supported:

specification file The name of a file listing the input rooted weak realizations.

EXAMPLES

Example 1: To compute a strong realization based on the realizations listed in "specfile":

```
example%  tree_merge specfile
```

Example 2: To compute a strong realization containing the realizations in the files listed:

```
example%  tree_merge –u <<!
            Groucho groucho.top
             Harpo harpo.top
            Chico chico.top
           !
```

ENVIRONMENT VARIABLES

All ignored.

EXIT STATUS

0 All information was written successfully.

>0 An error occurred.

BUGS

Program still under development, it seems to have problems with some test cases.

SEE ALSO

weak, realize

AUTHOR David F. Shallcross

NOTES

Path information collected from multiple starting hosts must be merged manually (for now).

A.28. weak

NAME

weak – compute a weak or rooted weak realization of a distance matrix

SYNOPSIS

felix/realizations/weak [-s | -u][-r <root name> | -rall <directory name>] [<matrix file name>]

OPERATING SYSTEM

Sun Solaris UNIX version 7 (gcc compiler)

DESCRIPTION

A weak realization of a distance matrix is defined to be a connected graph, with edge lengths, so that the distances between specified nodes of the graph are greater than or equal to the corresponding entries in the matrix. A minimum total length weak realization is necessarily a tree. A rooted weak realization is a weak realization for which the distances to a particular node, the root, are equal to the corresponding entries in the matrix. A minimum total length rooted weak realization will also be a tree.

This program reads a distance matrix, in the same format as fxtree, realize, and process_mat, assumed to be symmetric, nonnegative, and obedient to the triangle inequality. Depending on the options specified, it either looks for a minimum total length weak realization, for a minimum total length weak realization rooted at a specified monitor, or for a minimum total length rooted weak realization for each monitor. In the first two cases the resulting graph is written to standard output as a topology file. In the final case, each realization is written as a topology file with a name based on the monitor name, in the specified directory. A file called "specfile" will also be written in that directory, giving roots and file names of the rooted weak realizations.

Options:

-r <root name>   produce a realization rooted at the monitor named <root name>. If the delay matrix doesn't have monitor names, interpret <root name> as a number, starting from 0.

-rall <directory> produce a realization rooted at each monitor, and write each out to its own file in the directory given (which must pre-exist). Also write out a file "specfile" in that directory, consisting of lines giving the root monitor name and the name of the topology file for that realization, relative to the current default directory.

-s   write two link records for each edge in the realization. This is the default.

-u   write only one link record for each edge in the realization. For rooted realizations, this link will be oriented away from the root.

OPERANDS

The first argument without a leading dash will be the name of the matrix file. If there is none, the matrix will be read from standard input.

EXAMPLES

Example 1: To read in a matrix file "delay.mat", create a weak realization, and write it to standard output with two link records for each edge.

```
example%  weak "delay.mat"
```

Example 2: To read in a matrix from standard input, create a weak realization rooted at the monitor named "Harpo", and write it to standard output with one link record for each edge, oriented away from the root..

```
example%  weak –r Harpo –u
```

ENVIRONMENT VARIABLES

All are ignored.

EXIT STATUS

0	All information was written successfully.
>0	An error occurred.

BUGS

None known.

SEE ALSO

realize, process_mat

AUTHOR   David F. Shallcross

A.29. Template for Felix Program Manual Page

NAME

command name - description

SYNOPSIS

path [options]

alternate version path [options]

OPERATING SYSTEM

DESCRIPTION

    Detailed text description
    option1   description
    option2   description, etc

 PERMISSIONS

OPERANDS

    The following operand is supported:
    file   A path name of a file to be written.

USAGE

EXAMPLES

    Example 1: Using the XX command.
    example% XX -x
                    What's accomplished

ENVIRONMENT VARIABLES

EXIT STATUS

    0   All information was written successfully.
    >0   An error occurred.

BUGS

FILES   path/file

SEE ALSO   path/exe

AUTHOR

NOTES

Appendix B. Detailed documentation of standard file formats

Guidelines for X,Y files and topology files

1. Comments begin with an octocet ('#') and continue to the end-of-line
2. Blank lines and comments are ignored
3. Multiple consecutive whitespace characters (blanks or tabs) count as a a single separator between items.
4. Each content line begins with a single keyword, which defines the Information presented on that line and the format in which it is given.
5. A keyword ends with a colon (':') with no whitespace between the colon and the preceding alphabetic characters, and at least one whitespace after the colon.
6. No format or individual item can extend across more than one line; for lists of items (such as the list of monitor names or the list of links), the keyword may appear on multiple lines, each of which adds more entries to the total list.
7. Keywords are case-insensitive; recommended capitalization as shown below
8. The first three content lines must be "FileType:", "Version:", and "Generated:", in that order.

B.1. Topology File Formats

```
FileType: Felix Network Topology File
Version: 1.0
Generated: <by who/what, when, etc>

NumberOfMonitors: 3      # An integer, no smaller than 2
NumberOfNodes: 1         # An integer no smaller than 1
NumberOfLinks: 6         # An integer no smaller than (2 * NumberOfMonitors) +

MonitorNames: A B C # arbitrarily many names, separated by whitespace
NodeNames: X # arbitrarily many names, separated by whitespace
Link: linkname source dest bandwidth fixdelay del-dist numprm paramlst…
Link: %s       %s   %s   %lf     %lf     %s     %d     %lf …
Forward: vertexname destinationMonitor linkname
# If destinationMonitor is *, then all possible monitors for which no forwarding entry has so far been
# defined at this vertex will be routed out on the specified link.
# Forwarding table entries: these are not named, but simply list the next hop (identified by the link to use)
# over which a vertex sends all packets ultimately destined to a particular monitor.
```

B.2. Felix XY File format

```
FileType:    Felix X-Y Data File
Version:     1.0
Generated:   adxy2felixxy (jjd) on Wed Jul 14 18:58:24 1999
X-Units:     seconds
Y-Units:     milliseconds
X-Base:      0
Y-Base:      0

# time                        delay
# (sec.usec. since 1970)   (millisec.)
```

B.3. Postgres Database format

Field	Type	Length (bytes)
ts_sec	int4	4
ts_usec	int4	4
src	varchar()	16
dst	varchar()	16
starttime	varchar()	30
endtime	varchar()	30
delay	int4	4
pktloss	int4	4
pktsrcv	int4	4
bw	int4	4
ver	varchar()	30

B.4. Felix Monitor Names Mapping File

The Felix Monitor Names Mapping File provides the mapping between host/router names, host/router IP addresses, and symbol names. It may also include location information for the hosts/routers. The header for the file consists of the following lines:

Filetype:Felix_Monitor_Names_Mapping_File

Version: 1.0

Generated: <*optional information describing how the file was generated*>

Dataset: <*name of data set*>

After the header information, each line of the file describes one host or router. The hosts/routers may be monitors or may be internal nodes. The format of this description is as follows:

<*symbol*> <*IP address*> <*host name*> [<*host alias*>]*
[(@ | #) <*x coordinate*> <*y coordinate*>]

If a symbol mapping line ends with "@ <*x coordinate*> <*y coordinate*>", then the specified X and Y coordinates are used as the location for the node and are interpreted as such by the symbol file reading routines (used by several of the Felix tools). However, if the line ends with "# <*x coordinate*> <*y coordinate*>" then the X and Y coordinates are treated as a comment and ignored. The aliases and coordinate information are optional. Lines beginning with '#' are treated as comments.

Example mapping files are shown in Figure 41 and

Figure 42.

Felix Monitor Names Mapping Files are generated by the *pathconvert* program or by hand. They are used by *pathconvert*, *fxplot*, and other programs.

```
Filetype:  Felix_Monitor_Names_Mapping_File
Version:  1.0
Generated:            commonlinks (bss) on Thu Sep 24 09:58:40 1998
Dataset:  dataset5

#symbol  address             name
A        128.96.73.102       buzzard
B        128.96.73.75        santorini
C        192.4.16.66         offkey
D        207.3.231.32        bluemoon
E        192.4.18.61         brooklyn
F        192.4.6.9           breeze
G        192.4.13.8          wind
n001     128.96.73.254       128.96.73.254
n002     128.96.215.253      128.96.215.253
n003     128.96.9.4          128.96.9.4
n004     128.96.34.10        optimator.cc.bellcore.com
n005     128.96.34.15        celebrator.cc.bellcore.com
n006     128.96.63.10        duluth
n007     192.4.5.9           breeze-fddi
n008     128.96.9.2          128.96.9.2
n009     192.4.16.1          optimator
n010     128.96.34.254       mrehub-cisco.cc.bellcore.com
n011     128.96.9.3          128.96.9.3
n012     128.96.215.254      128.96.215.254
n013     128.96.9.1          128.96.9.1
n014     207.3.231.1         duluth
n015     128.96.63.1         optimator
n016     192.4.18.15         celebrator
n017     192.4.18.1          optimator
n018     192.4.5.1           optimator
n019     192.4.5.15          celebrator
n020     192.4.13.23         celebrator
```

Figure 41. dataset5.sym - Monitor Names Mapping File for Felix Dataset 5.

```
Filetype:  Felix_Monitor_Names_Mapping_File
Version:  1.0
Generated:            Bruce Siegell (bss) on July 14, 1999.
Dataset:  surveyor

#symbol  address             name
arl                 192.12.65.65        arl.dren @ -076.978693 38.988903
cmu                 128.2.3.101         cmu.csg ippm.net.cmu.edu @ -079.943027 \
         40.444451
colorado 204.131.62.126      colorado.csg \
         colorado.mm-ippm.advanced.org @ -105.360500 40.033600
harvard             128.103.160.162     harvard.csg @ -071.125253 \ 42.377421
umn                 160.94.54.248       umn.csg ippm-surveyor.nts.umn.edu \ @ -
         093.232209 44.972845
utah                155.99.55.11        utah.csg ippm.utah.edu @ \
         -111.852799 40.764689
virginia 128.143.99.21       virginia.csg \
         bootp-99-21.bootp.Virginia.EDU @ -078.470400 37.998500
washington          140.142.16.227      washington.csg \ ippm.cac.washington.edu @ -
         122.310723 47.653233
wisc                192.150.1.22        wisc.csg @ -089.406728 43.075090
```

Figure 42. surveyor.sym – Monitor Names Mapping File for Surveyor data set. Latitudes and longitudes are included.

B.5. Felix Path File

The path file describes paths from source monitors to a set of destination monitors. The header for the file consists of the following lines:

```
Filetype:Felix Path File
Version:        1.0
Generated:      <optional information describing how the file was generated>
Paths:          <number of paths listed in the file>
```

After the header information, each line of the file describes one path. The format of this description is as follows:

<source name> <destination name> : *<source IP address>*
[<intermediate hop IP address>] <destination IP address>*

Lines beginning with '#' are treated as comments.

An example of the Felix Path File is shown in ...

Felix Path Files are generated by the *tracepath* program and used by the *pathconvert* program. We have also written ad hoc tools to extract the path information from the Surveyor data and to convert the information into a Felix Path File.

Appendix C. Detailed documentation of protocols

C.1. Monitor Data Exchange Protocol

The MDEP has three functions:

Collecting delay and loss rates observed between two monitors

Learning the identity of other monitors in the network.

Broadcasting the information collected by the monitors to all elements of the monitoring infrastructure.

These functions are performed by exchanging status messages at random intervals. The status messages are UDP packets that carry the following information.

Source Address	Sending monitor
Source Port	Sending Process
Destination Address	Another monitor in the network
Destination Port	Port associated with process in the other monitor.
Message Type	Allows for future versions
Signature	MD5 signature of the message
Incarnation Number	Identifier of an instance of a monitor
Sequence Number	Identifier for a source/destination pair
Date, time	Sending date and time of the packet
Peer Incarnation Number	Last incarnation number received from that peer.
Peer Sequence Number	Last serial number received from that peer.
Last date and time indicator	The value indicated in the last message
Received Date, Time	Date, time when the last message was received
Per monitor Information	A list of observations describing the delays and loss rates of the network monitors

The per monitor information describes the quality of the reception from a specific monitor. For each monitor, the following information will be included.

Monitor Address	IP address of the monitor
Monitor Port	Port of the monitoring process
Loss Rate	Observed between the reporting monitor and that monitor.
Delay	Observed between the reporting monitor and that monitor.
Throughput	To be implemented in the next version

Each message will not necessarily provide information on all monitors in the network. This is to ensure that each message fits within a single UDP datagram, without fragmentation. The monitors described in any message are picked at random among the set of monitors.

The security information (MD5 digest) assumes that there is a globally assigned password to the monitoring network. The mechanism for managing this aspect is the following

prepare the complete message, initialize the security field to the password value

compute the MD5 checksum over the entire message

replace the password with the message digest computed above.

The converse is used to verify the message upon reception.

Messages are sent at irregular, pseudo-periodic intervals to avoid network overload and synchronization. At each interval the monitor to which a message will be sent will be chosen at random from the set of monitors.

C.2. Monitor to Archive Server Packet Format

Per monitor information is composed of the following fields:

2 bytes	pkt length
4 bytes	rcvtime sec
4 bytes	rcvtime usec
4 bytes	dst_ip_addr
2 bytes	dst_port
4 bytes	src_addr
2 bytes	src_port
4 bytes	msg type
4 bytes	sendtime sec
4 bytes	sendtime usec

append the monitor to monitor pkt format to the end of this pkt.

C.3. GUI to archive server packet format

Format of Request Pkt:

```
     <Type Of Graph - int>
    <Type of Stat  - int>
    <Src Monitor   - String>
    <Dst. Monitor  - String>
    <StartTime     - String>
    <End Time      - String>
    <DB filename   - String>      allows use of simulated DB or actual DB
    <user id.      - String>
<Parameter     - String(1 or more)>
    Parameter String:
     Topology then Parameter        ::=  <algorithm id>
     Prob Density Func then Parameter::=  <# of bins>
     Autocorrelation then Parameter ::=  <max time lag>
     Log Log Distrbtn then Parameter::=  <# of bins>
     Variance Time                  ::=  <Max. block size>
     Stats Chart                    ::=  digit 0
     Time Series then Parameter     ::=  <subsmple factor><smooth blk size>
                                    <overlap factor>
    Graph Type ::= [0 - Time Series | 1 - PDF | 2 - Autocorrelation |
                   3 - Log.Log.Distribution  | 4 - Topology |
                   5 - Time Variance        | 6 - Stats Chart]
    Stat Type  ::= [0 - Delay | 1 - Loss | 2 - Throughput]
```

References

F. Chung, M. Garrett, R. Graham, D. Shallcross, “Distance realization problems with applications to Internet tomography, submitted for publication, J. Computer and System Sciences, Feb 2000.

E.W. Zegura, K. L. Calvert, S. Bhattacharjee, “How to model an internetwork”, Proc. IEEE Infocomm, San Francisco, Sept 1996, pp 40-52.

[Cal97] K. L. Calvert, E. W. Zegura, M. Doar, "Modeling Internet topology", IEEE Commun. Mag., June 1997.

Andreas Fasbender and Ingo Rulands, "On Assessing Unidirectional Latencies in Packet-Switched Networks", Proc. ICC '97 pp. 490-494, 1997.

M. W. Garrett, "Inferential methods for network topology autodiscovery", written for Telcordia Exchange Magazine, July 2000.

M. W. Garrett, J. DesMarais, C. Huitema, P. Seymour, D. Shallcross, B. Siegell, "Felix Project: Network Topology Discovery and Performance Estimation from One-way Delay Measurements", in preparation, July 2000.

M. W. Garrett, J. Baron, P. Seymour, D. Shallcross, "Network Topology Discovery from Cross-correlation of Packet Delay Measurements", in preparation, July 2000.

M. Mathis, J. Mahdavi, "Diagnosing Internet Congestion with a Transport Layer Performance Tool", Proc. INET'96, June 1996.

M. Mathis, J. Mahdavi, G. L. Huntoon, V. Paxson, "Creating a National Internet Measurement Infrastructure", proposal to NSF (extending existing NLANR contract), Jan 1997.

D. Mills, "Adaptive Hybrid Clock Discipline Algorithm for the Network Time Protocol", ACM/IEEE Trans. Networking, pp. 505-514, Oct 1998.

R. Caceres, N. Duffield, S. B. Moon, D. Towsley. "Inference of Internal Loss Rates in the MBone," to appear, Global Internet Symposium '99.

[Moon99] S. B. Moon, P. Skelly, D. Towsley. "Estimation and Removal of Clock Skew from Network Delay Measurements," Proceedings of 1999 IEEE INFOCOM, New York, NY, March 1999.

[Moon00] Sue B. Moon, “Measurement and Analysis of End-to-end Delay and Loss in the Internet”, Ph.D. Thesis, Univ. Mass., Amherst, Feb. 2000.

Vern Paxson, "On Calibrating Measurements of Packet Transit Times", Proc. SIGMETRICS, 1998.

V. Paxson, J. Mahdavi, A. Adams, M. Mathis, "An Architecture for Large-Scale Internet Measurement", IEEE Comm Mag, pp. 48-54, August 1998.

[Ratn00] S. Ratnasamy, S. McCanne, “Inference of Multicast Routing Trees and Bottleneck Bandwidths using End-to-end Measurements”, IEEE Infocom, New York, March 1999.

D. Rubenstein, J. Kurose, D. Towsley, “Detecting Shared Congestion of Flows via End-to-end Measurement”, ACM Sigmetrics, June 2000.

D. Shallcross, M. W. Garrett, J. Baron, J. DesMarais, P. Seymour, B. Siegell, "Felix Project: Inference of Network Topology from Packet Delay Measurements", SIAM Annual meeting, Puerto Rico, July 2000.

B. Siegell, J. DesMarais, M. Garrett, P. Seymour, D. Shallcross, “Felix Project: Topology Discovery from One-way Delay Measurements" Passive & Active Measurement Workshop, University of Waikato, Hamilton, New Zealand, April 2000.

C. Tebaldi, "Bayesian inference on network traffic using link count data", JASA Theory and Methods, Vol 93, No 442, pp 557-76, 1998.

Y. Vardi, "Network Tomography: estimating source-destination traffic intensities from link data", JASA Theory and Methods", Vol 91, No 433, pp 365-77, 1996.

M. Yajnik, J. Kurose, D. Towsley, "Packet loss correlation in the MBone multicast network', Proc. IEEE Global Internet, Nov 1996.

E. W. Zegura, K. L. Calvert, M. H. Donahoo, "A quantitative comparison of graph-based models for internet topology", IEEE Trans. Networking, Vol 5, No 6, Dec 1997.

[SNMP96] SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M., and S. Waldbusser, "Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1902, IETF, January 1996. (See also RFCs 1903-7, 1155, 1157, 1212.)